An Exact Solver for the Minimum Line Cover of Prime-Indexed Points

Jesper Gran Mikkelsen
Independent Researcher, Norway
jesgranbusiness@gmail.com

May 2026

Abstract

Place the N prime points $(1, p_1), (2, p_2), \ldots, (N, p_N)$ in the plane, where p_k denotes the k-th prime number. The minimum number of straight lines needed to cover all N points, denoted $f(N)$, defines the integer sequence OEIS A373813. Every line through three or more prime points encodes an exact arithmetic coincidence; as N grows, such coincidences become vanishingly rare, making $f(N)$ computationally hard to certify exactly.

In this paper, we certify $f(1024) = 143$ as the new computational record, extending the prior certified boundary from $N = 861$ to $N = 1024$ with 163 newly certified, strictly optimal terms. While the previous record required approximately 282 hours using a general-purpose mixed-integer programming (MIP) solver [2, 3], our specialized C++ solver, `primecover1024.cpp`, reaches the $N = 861$ milestone in approximately 22 minutes and completes the full sweep to $N = 1024$ in under 40 hours on a Google Cloud `c4d-highcpu-8` instance (8 vCPUs, 15 GB RAM).

The solver performs an incremental sweep, enumerating all 12,162 *heavy lines* (those through at least three prime points) at startup in 50 ms, each stored as a 1024-bit coverage mask. At each step, it carries forward a primal warm-start solution, a Lagrangian dual seed, and a *witness cover* from $f(N-1)$. If the witness already covers the new point (N, p_N), optimality is certified instantly. Otherwise, a cover-inequality floor $\lceil |S|/2 \rceil$—derived from a greedy 2-cover-free subset S with no active heavy line covering three or more members—is established once at the root before branching begins. Exact branch-and-bound is then invoked: per-node pruning uses a *Lagrangian gain bound* tightened via projected subgradient descent on per-point dual variables $y \in [0, 1]^{U^+(T)}$, followed by a coordinate-descent polish pass when the bound falls within a fixed window of the pruning threshold.

Efficiency is further enhanced by the *Exclusive Dependency Rule*, which forces into the partial solution any *productive* heavy line for which at least three uncovered points each lie on no other *productive* heavy line—without loss of optimality. When the lower bound is within one of the incumbent and at least 64 points remain uncovered, strong branching evaluates a shortlist of at most four candidate lines via Lagrangian simulation. Independent subproblems are dispatched to worker threads via a shared atomic task queue. Altogether, we identify 20 new *awkward primes*—values of N at which $f(N)$ strictly exceeds $f(N-1)$—spanning $N = 864$ through $N = 1015$, with cover sizes ranging from 124 to 143 lines. Because the 1024-bit mask is the architectural limit (`kBitCapacity = 1024`), extending the record beyond $N = 1024$ requires a wider solver variant.

MSC 2020 Subject Classification: 05B40, 11A41, 68W40, 90C57.

Keywords: prime points, minimum line cover, set cover, parallel search, branch-and-bound, Lagrangian relaxation, cover inequalities, witness propagation, incremental sweep, awkward primes, OEIS A373813, OEIS A393445.

1 Introduction

The problem

Given the N points $(1, p_1), (2, p_2), \ldots, (N, p_N)$ in the plane, where p_i denotes the i-th prime, what is the minimum number of straight lines needed to pass through all N points simultaneously? Write

$$\mathcal{S}_N \;\coloneqq\; \{(1, p_1),\, (2, p_2),\, \ldots,\, (N, p_N)\}.$$

Three or more points of $\mathcal{S}_N$ are collinear only when their prime values satisfy an exact arithmetic relation placing them on a common rational line. Let $f(N)$ denote the minimum number of lines needed to cover $\mathcal{S}_N$; these counts form OEIS sequence A373813 [3]. The sequence is non-decreasing and almost everywhere flat: $f(N) = f(N-1)$ for the overwhelming majority of indices, because almost every new prime falls on a line already present in an optimal cover. The exceptional indices—those at which $f(N) > f(N-1)$—are called *awkward primes* (OEIS A393445 [4]), and certifying a new record requires proving optimality at every one of them up to the new boundary.

Prior work

The prior certified record stood at $N = 861$ [2], reached using an industrial MIP solver in 282 hours, 26 minutes, and 31.5 seconds of compute time. General-purpose MIP solvers treat each value of N as an independent set-cover instance and cannot exploit the arithmetic structure of prime collinearity or the warm state built up by earlier instances. That computation serves as the external benchmark throughout this paper.

Contribution

This paper describes `primecover1024.cpp`, a purpose-built exact solver that replaces the MIP approach with an incremental, arithmetic-aware architecture. Two phases structure the computation.

Phase 1: heavy-line enumeration (≈ 50 ms, one-time) A *heavy line* is any line through at least three points of $\mathcal{S}_{N_{\text{end}}}$. The solver enumerates all heavy lines once before the sweep begins, storing each as a 1024-bit point-coverage bitmask (sixteen 64-bit words, occupying 128 bytes).

A heavy line becomes *active* at step N only once its third-smallest point index has been reached; before that it cannot improve any optimal cover. Because all per-step intersection counting reduces to popcount operations on these fixed-width bitmasks, the core working data fits entirely in the L1/L2 cache throughout the sweep.

Phase 2: incremental sweep (under 40 hours end-to-end) The solver processes $N = 1, 2, \ldots, N_{\text{end}}$ in order, carrying three warm pieces of state across each step: a *witness cover* (an explicit set of lines that certifies $f(N-1)$, extended by a greedy look-ahead that pairs residual uncovered points to maximise coverage of future indices up to N_{end}); a *warm primal seed* (the optimal heavy-line selection from the previous step); and a *warm Lagrangian dual seed* (the dual multiplier vector from the previous Lagrangian solve). Each step resolves in one of three modes.

Witness mode (`W`). If the new point (N, p_N) already lies on a line in the current witness cover, the answer $f(N) = f(N-1)$ is inherited without any search.

Root mode (`R`). If the witness fails, the solver computes two lower bounds before any branching. The *cover-inequality lower bound* $\lceil |S|/2 \rceil$ is derived from the largest subset $S \subseteq \mathcal{S}_N$

such that no active heavy line covers three or more points of S; since every line covers at most two members of S, at least $\lceil |S|/2 \rceil$ lines are needed. A *Lagrangian gain bound*—an upper bound on the total gain attainable by selecting further active heavy lines, and therefore a lower bound on $f(N)$—is then computed at the root via projected subgradient descent on dual variables $y \in [0,1]^N$. If the tighter of these two bounds matches the best known feasible cost, the answer is certified without branching.

Exact mode (`D`). If a gap remains after the root bounds, the solver runs branch-and-bound warm-started from the primal and dual seeds. At every node the *Exclusive Dependency Rule* forces into the partial solution any *productive heavy line*—an active heavy line currently covering three or more uncovered points—for which at least three uncovered points lie on no other productive heavy line; an exchange argument (theorem 6.3) proves this never worsens the solution. The Lagrangian gain bound is recomputed at every node, and a coordinate-descent polish pass is applied when the node bound falls within a fixed window of the pruning threshold. When the gap between the node lower bound and the incumbent is at most one and at least 64 uncovered points remain, *strong branching* evaluates a shortlist of at most four candidate lines—each assessed by a Lagrangian simulation of the include branch—and selects the variable that maximises combined child-node pruning. Once the root-level work is complete, independent subproblems from a pre-expanded frontier are dispatched to all available hardware threads.

The solver closes the gap between its combined lower bound and its best incumbent to zero before terminating; it has no heuristic stopping rule.

The concrete outcomes are as follows.

- **New record.** $N = 1024$ is certified optimal, with $f(1024) = 143$. The full sweep from $N = 1$ through $N = 1024$ completes in under 40 hours on a Google Cloud `c4d-highcpu-8` instance (8 vCPUs, 15 GB RAM). On the same hardware, the sweep surpasses the prior boundary of $N = 861$ in approximately 22 minutes.
- **Sequence extension.** A373813 is extended by 163 newly certified terms, from $N = 862$ through $N = 1024$.
- **New awkward primes.** The 163-term extension contains 20 new awkward primes (extending OEIS A393445 [4]), listed in table 1 immediately below.
- **Optimality guarantee.** The reported $f(N)$ is proven optimal for every reported N; the algorithm never relies on a heuristic termination criterion.

Table 1: New awkward primes in the range $N = 862$–1024, with the prime p_N and the new minimum line count $f(N)$ at each awkward index.

N	p_N	$f(N)$	N	p_N	$f(N)$
864	6701	124	934	7331	134
871	6763	125	941	7417	135
875	6793	126	944	7457	136
883	6863	127	948	7487	137
893	6959	128	957	7547	138
895	6967	129	965	7591	139
908	7069	130	978	7703	140
918	7187	131	981	7727	141
922	7213	132	990	7829	142
928	7253	133	1015	8081	143

Computational complexity

The minimum line cover problem—given an arbitrary finite point set P in the plane and a positive integer k, do k lines suffice to cover P?—is NP-complete; it is a geometrically structured special case of the classical *set cover* problem, and the prime-point instances $\mathcal{S}_N$ are instances of this general problem. Megiddo and Tamir [1] establish hardness by reduction from 3-SAT, encoding each variable as a binary line-choice gadget and each clause as a point covered if and only if a satisfying literal's line is selected.

The branch-and-bound explores a binary include/exclude tree over productive heavy lines. Since a line h is active at step N only after its three smallest point indices are all in $\mathcal{S}_N$, every active heavy line is productive at the root, giving a worst-case tree of $2^{|A_N|}$ leaves, where A_N denotes the set of active lines at step N. Four mechanisms prune this space: (i) *Lagrangian bounding* eliminates subtrees whose gain upper bound cannot beat the incumbent (`lag_prune`); (ii) *the Exclusive Dependency Rule* (theorem 6.3) forces every productive line h for which at least three uncovered points each lie on no other productive line, reducing effective depth (`forced`); (iii) *dominance pruning* (theorem 6.4 and algorithm 3) discards frontier tasks whose state—the bitmask of uncovered points together with the set of blocked productive lines—has been reached at weakly higher accumulated gain; and (iv) *branch-specific child caps*, computed from the Lagrangian dual at the current node, tighten the bound for each branch direction individually. In practice the maximum DFS depth across the full sweep is 108 and node counts at the hardest instances reach the hundreds of millions, far below $2^{|A_N|}$ with $|A_N| \approx 9{,}000$–$12{,}000$ at the new awkward primes.

The sections mirror the solver's decomposition. Section 2 defines the geometric objects, the gain reformulation used throughout the solver, and the task state. Section 3 describes the two-phase architecture: heavy-line enumeration, the exact-search skeleton, and the parallel frontier mechanism. The three execution modes, witness construction and propagation, and the warm state carried between successive values of N are developed in section 4. Section 5 develops the cover-inequality lower bound, the Lagrangian gain bound, the projected subgradient and coordinate-descent polish steps, and the branch-specific child caps. Section 6 proves correctness, including the Exclusive Dependency Rule and the mode-by-mode certification guarantee. Section 7 records the new OEIS data and per-instance statistics extracted from the sweep log. Section 8 closes with code-availability notes and the precise meaning of "`1024`" in the filename.

Repository, Results & Interactive Demo

The solver source, certified results, and all supplementary material are collected in the GitHub repository below. The repository also includes a browser-based visualisation that runs the same bitmask branch-and-bound algorithm in a Web Worker, drawing each optimal cover as N increments from 1 upward in real time, with a speed control to pace the animation. The covers in Figures 1 and 2 are independent TikZ renderings from precomputed exact data; the browser demo conveys the geometric progression of the cover structure in a way that static figures cannot. The JavaScript implementation reaches $N \approx 300$ in eight seconds, while the C++ solver covers the same range in 568 ms.

Repository: https://github.com/jespergran98/prime-line-cover

Demo: https://prime-line-cover.vercel.app

2 Problem Definition

2.1 Prime Points and Heavy Lines

Write $[N] := \{1, 2, \ldots, N\}$ for the index set of the first N primes, where p_i denotes the i-th prime, and define

$$\mathcal{S}_N = \{(i, p_i) : i \in [N]\}, \qquad N \geq 1.$$

Three or more points of $\mathcal{S}_N$ are collinear only when their prime values satisfy an exact arithmetic relation—a coincidence that is sparse and arithmetically irregular. A line carrying $k \geq 3$ such points costs 1 in any cover, whereas covering the same k points with 2-point lines costs $\lceil k/2 \rceil \geq 2$, a saving of at least one line. Lines passing through at least three points of $\mathcal{S}_N$ therefore form the combinatorial skeleton of every optimal cover.

Definition 2.1 (Minimum line cover number)**.** *$f(N)$ denotes the minimum cardinality of a family of straight lines whose union contains $\mathcal{S}_N$.*

Equivalently, computing $f(N)$ is a *set cover* problem: the universe is $\mathcal{S}_N$, each candidate set corresponds to a line in the plane restricted to its intersection with $\mathcal{S}_N$, and the objective is to cover the universe with the fewest sets.

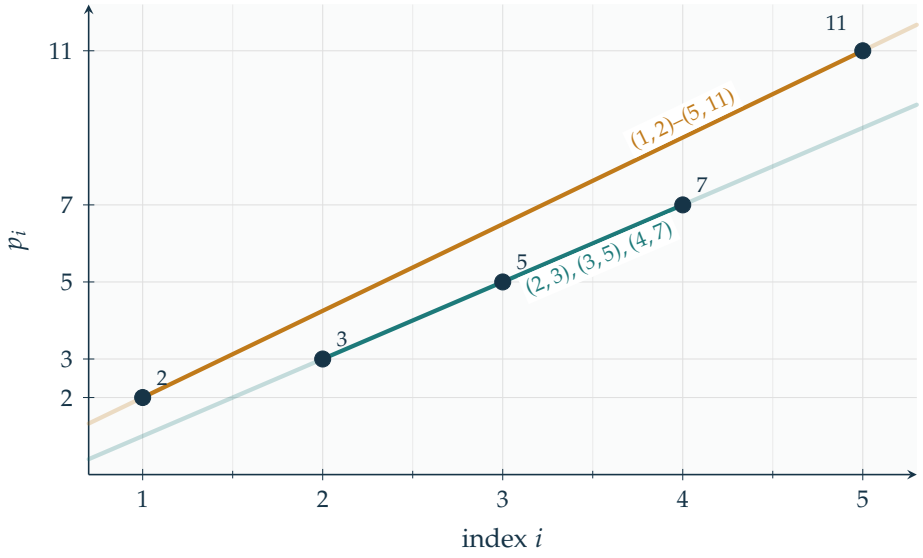

Figure 1: The prefix $\mathcal{S}_5$ is the last instance coverable by two lines. L_{pair} joins $(1, 2)$ to $(5, 11)$; L_{tri} passes through the three-point coincidence $(2, 3), (3, 5), (4, 7)$. Hence $f(5) = 2$, but the arrival of the prime 13 forces $f(6) = 3$.

Five lines still suffice to cover $\mathcal{S}_{24}$; in fact $N = 24$ is the last prefix for which $f(N) = 5$. Three of these five lines (L_A, L_B, L_C) are parallel with slope 4—an arithmetic coincidence among primes rather than a geometric inevitability. The five lines of an optimal cover, distinguished by line style in figure 2, support the following points of $\mathcal{S}_{24}$:

L_{pair}: $(1,2)$, $(24,89)$,
L_{short}: $(2,3)$, $(3,5)$,
L_A: $(4,7)$, $(5,11)$, $(17,59)$, $(19,67)$, $(20,71)$, $(22,79)$, $(23,83)$,
L_B: $(6,13)$, $(7,17)$, $(10,29)$, $(12,37)$, $(13,41)$, $(16,53)$, $(18,61)$, $(21,73)$,
L_C: $(8,19)$, $(9,23)$, $(11,31)$, $(14,43)$, $(15,47)$.

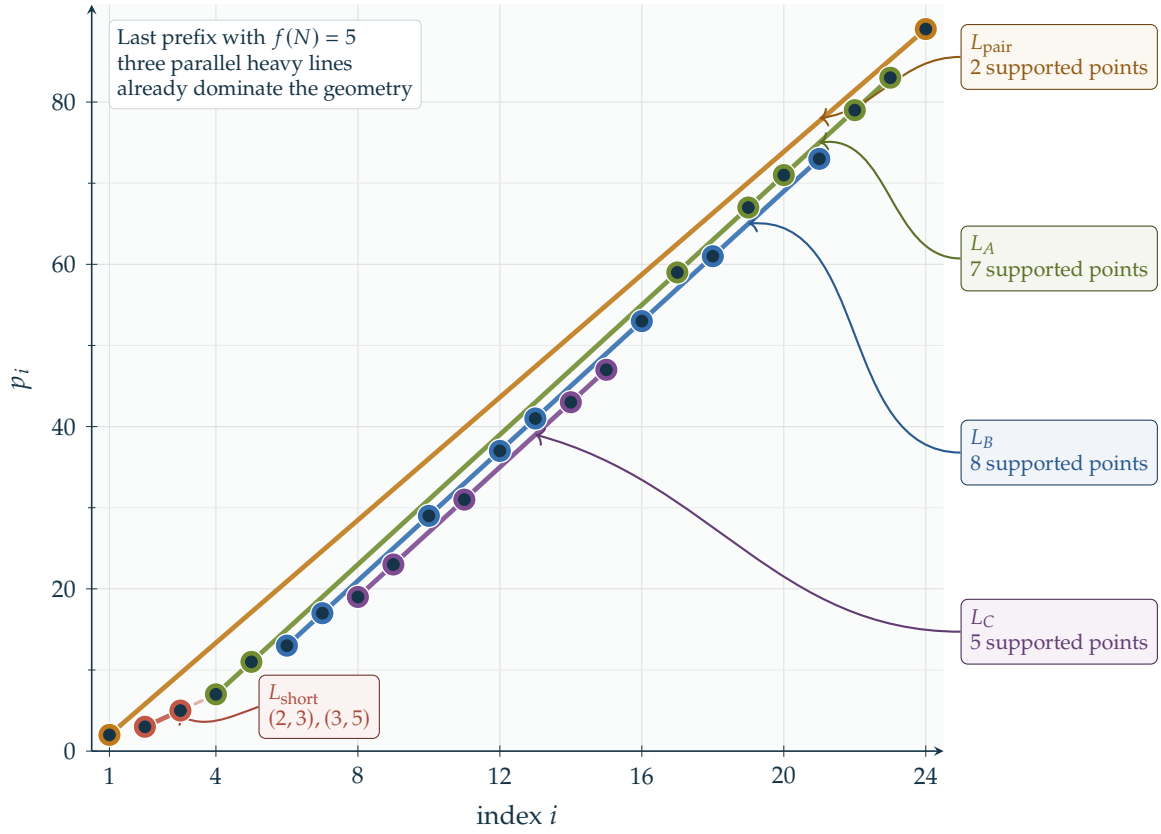

Figure 2: The dashed extension of L_{short} shows that the same geometric line passes through $(4,7)$, but that incidence is redundant since L_A already covers it. After $N = 24$, no cover of $\mathcal{S}_N$ can avoid a sixth line.

Lines are identified by integer arithmetic rather than floating-point coordinates, ensuring exact equality tests.

Definition 2.2 (Canonical line)**.** *Given two distinct integer-coordinate points (x_1, y_1) and (x_2, y_2), set*

$$a = y_1 - y_2, \qquad b = x_2 - x_1, \qquad c = -(ax_1 + by_1),$$

divide (a, b, c) by $g = \gcd(|a|, |b|, |c|)$, and negate the triple if $a < 0$, or if $a = 0$ and $b < 0$, so that $a > 0$, or $a = 0$ and $b > 0$. The resulting primitive triple (a, b, c) defines the canonical equation $ax + by + c = 0$.

Two pairs of integer points determine the same line if and only if they produce identical canonical triples, making line equality and hashing purely integer operations. This normalization is implemented by `canonical_line()` in the source.

Definition 2.3 (Heavy line). *A line is* heavy *if it contains at least three points of* $\mathcal{S}_{N_{\text{end}}}$. *The set of all heavy lines is* $\mathcal{H}$. *For* $h \in \mathcal{H}$, *write*

$$\text{pts}(h) = \{i_1 < i_2 < \cdots < i_t\} \subseteq [N_{\text{end}}]$$

for its point indices in increasing order.

Definition 2.4 (Activation index). *For* $h \in \mathcal{H}$ *with* $\text{pts}(h) = \{i_1 < i_2 < \cdots < i_t\}$, *define*

$$\text{act}(h) := i_3.$$

The set of active *heavy lines at step* N *is*

$$A_N := \{h \in \mathcal{H} : \text{act}(h) \le N\}.$$

Point indices are stored 0-based, so the code sets `activate_at = hl.points[2] + 1`, converting the third point's 0-based index to the 1-based $\text{act}(h)$. A heavy line enters `active_ids` at step $N = \text{act}(h)$, the first step at which $\mathcal{S}_N$ contains all three of the line's defining points.

2.2 Gain Reformulation

Once the heavy lines in some $H \subseteq A_N$ are chosen, every remaining uncovered point is handled by a 1- or 2-point line, at total residual cost $\lceil (N - |\text{cov}_N(H)|)/2 \rceil$, where $\text{cov}_N(H) := \bigcup_{h \in H} (\text{pts}(h) \cap [N])$. The combined cost is

$$|H| + \left\lceil \tfrac{N - |\text{cov}_N(H)|}{2} \right\rceil.$$

Definition 2.5 (Gain). *For* $H \subseteq A_N$, *the* gain *of* H *is*

$$\text{gain}_N(H) := |\text{cov}_N(H)| - 2|H|.$$

Proposition 2.6 (Cost–gain equivalence). *For every* $H \subseteq A_N$,

$$|H| + \left\lceil \frac{N - |\text{cov}_N(H)|}{2} \right\rceil = \left\lceil \frac{N - \text{gain}_N(H)}{2} \right\rceil.$$

In particular, minimizing the line count over all feasible covers is equivalent to maximizing $\text{gain}_N(H)$ *over all* $H \subseteq A_N$.

Proof. Substituting $\text{gain}_N(H) = |\text{cov}_N(H)| - 2|H|$ into the right-hand side gives

$$\left\lceil \frac{N - |\text{cov}_N(H)| + 2|H|}{2} \right\rceil = |H| + \left\lceil \frac{N - |\text{cov}_N(H)|}{2} \right\rceil,$$

which is the left-hand side. The equivalence of the two optimization problems follows because $g \mapsto \lceil (N - g)/2 \rceil$ is non-increasing. □

The solver searches over subsets of active heavy lines and tracks accumulated gain; the function `cost_from_gain(n, gain)` computes $\lceil (n - \text{gain})/2 \rceil$ as `(n - gain + 1) / 2` using integer arithmetic.

2.3 Search Nodes

The branch-and-bound search maintains a *task*: the state associated with a single search node. A task stores the following fields, indexed either by the 0-based heavy-line identifier `id` or by point index $p \in \{0, \ldots, N-1\}$:

- the *uncovered point set* $U(T) \subseteq [N]$, represented as a 1024-bit integer mask `task.active`; the companion integer `task.active_count` $= |U(T)|$ is maintained alongside for O(1) cardinality queries, and points are removed from both as they are covered by selected lines;
- for every active heavy line h, its *residual cover count*

$$c_T(h) := |\text{pts}(h) \cap U(T)|,$$

  stored as an unsigned 16-bit integer in `task.line_cover[id]`;
- an availability flag (`task.available[id]`), set to 1 if the line may still be selected and 0 once it has been selected or excluded;
- the heavy lines selected so far (`task.current_choice`) and their total accumulated gain (`task.current_gain`); and
- two auxiliary point-indexed arrays, `task.productive_degree[p]` and `task.sole_productive_line[p]`, recording, for each uncovered point p, the number of productive lines (defined below) that cover p and—when that count equals exactly one—the identity of that line. When the count is zero, `sole_productive_line[p]` holds the sentinel `kNoProductiveLine` ($= -1$); when the count exceeds one, it holds `kManyProductiveLines` ($= -2$). Both arrays are maintained incrementally and drive the Exclusive Dependency Rule described in section 6.

Definition 2.7 (Productive line)**.** *At node T, an available heavy line h is* productive *if $c_T(h) \geq 3$. The set of productive lines is tracked in the compact array* `task.productive_ids`*.*

Selecting a productive line h increases the accumulated gain by

$$\delta_T(h) := c_T(h) - 2 \;\geq\; 1,$$

the value returned by `positive_gain(task, id)`. Because the uncovered point set only shrinks as the search descends, a line whose residual coverage falls to 2 can never again contribute positive gain; it is removed from `task.productive_ids` as soon as $c_T(h)$ drops from 3 to 2.

Definition 2.8 (Blocked productive set)**.** *At node T, the* blocked productive set $B(T)$ *is the collection of* unavailable *heavy lines h with $c_T(h) \geq 3$, maintained in sorted order in* `task.blocked_productive_ids`*.*

A line enters $B(T)$ when it is excluded by a branching decision while its residual coverage is still three or more; it is evicted as soon as coverage drops below 3. Together with the uncovered-point mask, $B(T)$ forms the state key used for dominance pruning during frontier construction: a node at state $(U(T), B(T))$ is dominated and discarded if its accumulated gain does not exceed the maximum gain already recorded for that state.

3 Algorithm Overview

The solver proceeds in two phases. *Phase 1*, executed once at startup, identifies every *heavy line*—a line passing through at least three prime points (i, p_i)—and builds the index structures

Phase 2 requires. *Phase 2* sweeps $N = 1, 2, \ldots, N_{\text{end}}$ in order, maintaining a certified minimum cover size $f(N)$ at each step through an incremental exact search that reuses warm-start information from the previous step.

3.1 Phase 1: Heavy-Line Enumeration

For each *anchor index* i, algorithm 1 groups later indices $j > i$ by GCD-normalised slope; any slope bucket containing at least two such indices determines a line through three or more prime points. A candidate is emitted only when i is the leftmost prime index on the corresponding arithmetic progression, which is the sole de-duplication rule used throughout Phase 1.

Each emitted heavy line ℓ stores three structures built at emission time: a 1024-bit coverage mask with bit k set iff prime index k lies on ℓ; an ordered point list; and an activation step $\ell.\texttt{activate\_at} = a_3 + 1$, where a_3 is the *zero-based* index of the line's third-smallest prime point (equivalently, `points`[2] + 1 in the implementation). After enumeration the emitted lines are sorted by first-point index ascending, then point-set size descending, then lexicographic order.

Algorithm 1 | EnumerateHeavyLines($p_0, \ldots, p_{N_{\text{end}}-1}$)

for $i \leftarrow 0$ **to** $N_{\text{end}} - 1$ **do** — *anchor sweeps every point left-to-right*

- clear `slope_buckets` — *fresh grouping per anchor i*
- **for** $j \leftarrow i{+}1$ **to** $N_{\text{end}} - 1$ **do** — *pair anchor with every later point*
  - $sk \leftarrow$ reduced_slope$(j - i,\ p_j - p_i)$ — *GCD-normalised rational slope*
  - append j to `slope_buckets`[sk]
- **for each** bucket $B \in$ `slope_buckets` *with* $|B| \geq 2$ **do** — *≥ 3 points collinear with i*
  - *if i is the leftmost anchor index on this slope then* — *de-duplication: emit each line exactly once*
    - **emit** line ℓ through $\{i\} \cup B$
    - `activate_at`(ℓ) $\leftarrow$ `points`[2] + 1 — *zero-based index of third point, plus one*

Post-processing : sort emitted lines by **first-point index** ↑, then **point-set size** ↓, then **lexicographic order** ↑.

After enumeration the driver constructs two lookup structures that the sweep reuses at every step:

- an *incidence list* mapping each index $p < N_{\text{end}}$ to every heavy line whose point set contains p; and
- an *activation list* mapping each step N to the heavy lines whose `activate_at` equals N.

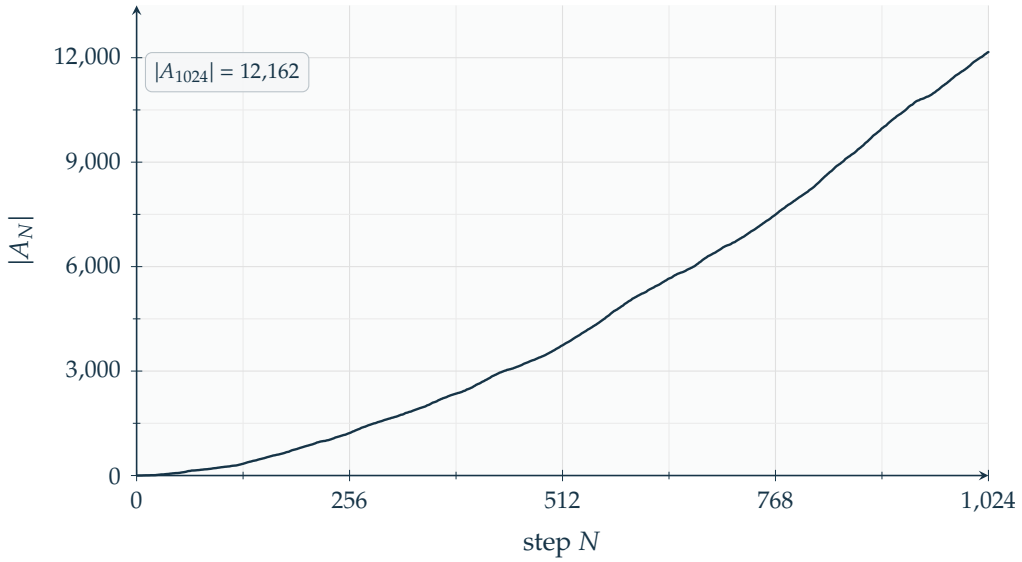

Figure 3: Number of active heavy lines $|A_N|$ at each step $N = 1, \ldots, 1024$. A line $h \in \mathcal{H}$ becomes active at step $N = \text{act}(h) = a_2 + 1$, where a_2 is the zero-based index of its third-smallest prime point (theorem 2.4); thus, $|A_N|$ represents the exact size of the solver's decision space at step N. The count grows from 0 at $N = 1$ to 12,162 at $N = 1024$, exhibiting a super-linear profile where new heavy lines activate at an accelerating rate as the sweep progresses. Each of these 12,162 masks occupies exactly 128 bytes (sixteen `uint64_t` words), so the total working set of roughly 1.5 MB remains L2-resident throughout the sweep regardless of $|A_N|$. The witness mechanism is equally critical: despite this monotonically growing decision space exceeding 10,000 active lines for the final hundred steps, the vast majority of those steps resolve in **Mode `W`** without invoking the exact solver, so the expanding decision space imposes no runtime penalty at non-awkward primes. The full set of 12,162 heavy lines is enumerated once before the sweep begins, in approximately 50 ms.

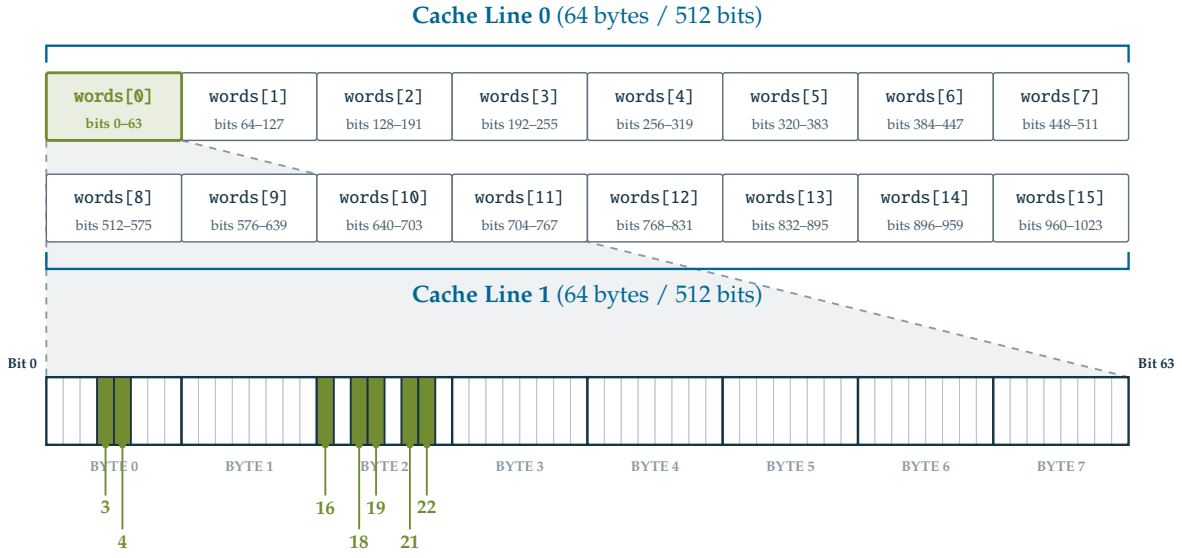

Figure 4: `BitMask1024` memory layout. Each of the 12,162 heavy lines is stored as sixteen contiguous `uint64_t` words (128 bytes, exactly two cache lines). Bit k encodes whether zero-based prime index k lies on the line, addressed as `words[`$k \gg 6$`]` at position k mod 64. Coverage counting at every branch-and-bound node reduces to `popcount` over word-wise AND, keeping the full working set in L1/L2 throughout the sweep. Filled cells show the bit pattern of line L_A from $\mathcal{S}_{24}$ (one of the five lines in the optimal cover shown in figure 2, covering 1-based prime indices 4, 5, 17, 19, 20, 22, 23, stored at zero-based bit positions 3, 4, 16, 18, 19, 21, 22); all set bits fall within `words[0]`.

3.2 Phase 2: Incremental Sweep

The sweep processes $N = 1, 2, \ldots, N_{\text{end}}$ in order. At step N the driver first appends to `active_ids` every heavy line whose `activate_at` equals N.

Witness mechanism. The solver maintains a *witness cover*: a set of lines whose union covers every prime point (t, p_t) for $t \le N_{\text{end}}$. If the witness carried from step $N-1$ already covers (N, p_N), the driver returns $f(N) = f(N-1)$ immediately, bypassing the exact solver (**Mode `W`**; see figure 5).

Greedy upper bound. At every step the driver calls `greedy_upper_bound_solution()`, which produces a feasible cover cost `ub_raw` by the following adaptive procedure: repeatedly select the active heavy line covering the greatest number of currently uncovered points, subject to that count exceeding 2, breaking ties in favour of the line with the larger total-point list; once no active heavy line covers more than two uncovered points, close the r remaining uncovered points with $\lceil r/2 \rceil$ lines. The driver then forms

$$\texttt{ub0} = \min\{N,\ \texttt{prev} + 1,\ \texttt{ub\_raw}\},$$

where `prev` is the last certified cover size, tightened to $\min\{\texttt{ub0}, \texttt{prev}\}$ when the carried witness already covers (N, p_N).

Exact solve. On a witness miss the driver calls `solve_exact()`. The driver first computes a scalar floor

$$\texttt{lb0\_floor} = \begin{cases} \texttt{prev}, & \text{if } \texttt{warm\_start\_solution} \text{ is nonempty,} \\ 0, & \text{otherwise,} \end{cases}$$

and passes it to `solve_exact()`, which independently computes the *cover-inequality lower bound* lb_{cov}—the maximum size of a subset S of the zero-based index set $\{0, \ldots, N-1\}$ such that no active heavy line covers more than two points of S, yielding $\lceil |S|/2 \rceil$ as a valid lower bound because every line covers at most two points of S (see section 5)—and passes $\max\{\texttt{lb0\_floor}, \text{lb}_{\text{cov}}\}$ as the combined floor to the gain solver. Inside the gain solver, `capture_root_cost_lb()` takes the maximum of this combined floor and the Lagrangian gain bound computed at the first evaluation—during frontier construction or the first DFS node—to produce the reported root lower bound lb (see section 5).

3.3 Exact Search and the Parallel Frontier

On a witness miss the solver constructs the root search task and establishes an initial incumbent through the following seeding sequence. First, `warm_start_solution` is replayed on a private copy of the root to generate a candidate; then `greedy_seed_line_ids` is replayed on a separate private copy for the same purpose; each result is submitted if it improves the current incumbent. The Exclusive Dependency Rule (algorithm 2) is then applied directly to the shared root, permanently forcing any provably required productive line. The solver records the productive-line count `prod` and total productive incidence `pinc` at this forced root state, which are tabulated in section 7. Two greedy completions then follow from the forced root: one without any seed, and one after replaying `warm_start_solution` onto a private copy of the forced root before completing greedily; each result is submitted as a candidate if it improves the incumbent.

Exclusive Dependency Rule. A *productive* line is a heavy line whose residual active-point count is at least three; its *gain* is $\text{gain}(h) := \texttt{line\_cover}[h] - 2 > 0$. Call an active point p *exclusively covered by* heavy line h if h is the only productive line through p. When h has at least three exclusively covered active points, any optimal solution that omits h can be restructured to include h without increasing cost (see proof below), so h may be forced into the partial solution unconditionally. The rule is applied iteratively: each round selects the qualifying line with the most exclusively covered active points, breaking ties first by larger gain and then by smaller line identifier, and repeats until no qualifying line remains.

Proof that the threshold is three. Let h be a productive line with $|E_T(h)| \geq 3$ exclusively covered active points, where $|E_T(h)|$ denotes the exclusive active-point count. Consider any optimal solution that omits h. Each point of $E_T(h)$ must be covered by a non-productive line, which covers at most two active points total; at most one of those two can lie outside $E_T(h)$. If k such lines are used and they collectively cover $d \leq k$ non-exclusive points, replacing them with h plus $\lceil d/2 \rceil$ pairing lines changes cost by $\delta = -k + 1 + \lceil k/2 \rceil$. For $k = 2$, $\delta = 0$; for $k = 3$, $\delta = 0$; for $k \geq 4$, $\delta < 0$. Because $|E_T(h)| \geq 3$ forces $k \geq 2$, we always have $\delta \leq 0$, so forcing h never increases cost. □

Algorithm 2 | ApplyExclusiveDependencyRule(T)

repeat *iterate until no qualifying productive line remains*

  for each $p \in U(T)$ **do** set $\texttt{sole}[p] \leftarrow$ unique productive line through p, or $\varnothing$ *$\varnothing$ if zero or ≥ 2 productive lines cover p*

  touched $\leftarrow \{\, \texttt{sole}[p] : p \in U(T), \texttt{sole}[p] \neq \varnothing \,\}$ *lines with ≥ 1 exclusively covered active point*

  best $\leftarrow \varnothing$; $u^* \leftarrow 2$; $g^* \leftarrow -1$ *threshold $u^* = 2$ enforces the $u \geq 3$ rule*

  for each $h \in$ *touched* **do** *score each candidate line*

    $u \leftarrow |\{p \in U(T) : \texttt{sole}[p] = h\}|$ *$|E_T(h)|$: exclusive active-point count*

    if $u < 3$ or $\text{gain}(h) \leq 0$ then **continue** *rule threshold and positivity guard*

    if $u > u^*$, or $\big(u = u^*$ and $\text{gain}(h) > g^*\big)$, or $\big(u = u^*$ and $\text{gain}(h) = g^*$ and $h <$ best$\big)$ then

      best $\leftarrow h$; $u^* \leftarrow u$; $g^* \leftarrow \text{gain}(h)$

  if best $= \varnothing$ then **break** *no productive line has $|E_T(\cdot)| \geq 3$ with positive gain; rule exhausted*

  force *best* into T *adds gain(best) to T.`current_gain`; updates $U(T)$ and `sole`*

Notation : $\text{gain}(h) := \texttt{line\_cover}[h] - 2$; $\texttt{sole}[p] = h$ iff h is the unique line in `productive_ids` through active point p; forcing *best* removes its active points from $U(T)$, which may create new exclusive dependencies in the next pass.

Bounds and search modes. If the root lower bound closes the gap, the solver exits without branch-and-bound (**Mode `R`**, zero DFS nodes counted); otherwise it builds a parallel frontier and launches depth-first search (**Mode `D`**). The root lower bound is the maximum of three quantities: the cover-inequality bound lb_{cov}, the Lagrangian gain bound evaluated at the forced root state, and the previous step's certified optimal cost (which serves as a warm lower-bound floor whenever a primal warm solution is available); all three are described in section 5.

Parallelism. If the machine reports a single hardware thread the solver runs depth-first search on the forced root task directly. Otherwise it first builds a parallel frontier whose target size is

$$\max\{1,\ \texttt{workers} \times m(C^*)\},$$

where C^* is the incumbent cover cost after root seeding and

$$m(C^*) = \begin{cases} 8192, & \text{if } C^* \geq 128, \\ 2048, & \text{if } C^* \geq 125, \\ 512, & \text{if } C^* \geq 121, \\ 128, & \text{if } C^* \geq 113, \\ 16, & \text{if } C^* \geq 93, \\ 4, & \text{otherwise.} \end{cases}$$

To expand the frontier the solver repeatedly extracts the task with the largest active-point count (ties broken by smaller current gain), applies the Exclusive Dependency Rule with a full metadata rebuild, computes the Lagrangian bound, branches on one productive line, and applies incremental forcing to each surviving child before inserting it into a *dominance map* keyed by $\text{key}(T) := \big(U(T), B(T)\big)$, where $U(T)$ is the 1024-bit active-point mask and $B(T)$ is the sorted list of unavailable lines whose residual coverage is at least three (theorem 2.8). Each child is discarded if its key already maps to a gain at least as large as the child's accumulated gain; otherwise the map is updated and the child is enqueued. The extracted task itself is discarded (without updating the map) only if the map already records a *strictly* greater gain for the same key, since equal-gain siblings represent genuinely distinct search paths that may still yield a better solution via different branching sequences. Before dispatch the frontier is sorted in descending order of `active_count`, then descending `productive_ids` size, then ascending `current_gain`, so that the hardest, least-pruned subproblems reach workers first. If the frontier reduces to a single task it is processed on the main thread; otherwise worker threads pull tasks via an atomic counter. Each worker maintains its own complete search context—Lagrangian scratch space, undo logs, and candidate buffers—so no locking is required on the DFS hot path. The only shared search state on the hot path is the atomic incumbent gain; a separate mutex serialises best-solution updates and end-of-task statistics aggregation, neither of which occurs on the critical path. These multiplier values were calibrated for the reference machine (`c4d-highcpu-8`, 15 GB RAM); the safe maximum on other hardware depends on per-task memory, which shrinks as the frontier grows and should be measured rather than estimated linearly (see section 8).

Branch selection. The solver forms a shortlist of up to four productive lines, ranked by larger residual gain $c_T(h) - 2$, then by larger non-negative Lagrangian slack, then by the branch-pressure surrogate

$$\sum_{p \in \text{pts}(h) \cap U(T)} |\texttt{ordered\_point\_lines}[p]|,$$

and finally by smaller line identifier; the exclude-child Lagrangian bound is computed for each shortlisted candidate. When the shortlist contains more than one line, the gap to the incumbent is at most one, the active-point count is at least 64, and the current number of explicit branching decisions on the path is at most six (tracked in `task.current_choice`, which does not include lines forced by the Exclusive Dependency Rule), the solver performs *strong branching*: each shortlisted line's include-child is evaluated with at most 24 Lagrangian iterations, and the winner is chosen by (i) most children pruned, (ii) smallest worst-child bound, (iii) smallest sum of child bounds, then (iv) the ranking rule above. Otherwise the solver branches on the top-ranked shortlist line.

Algorithm 3 | BUILDFRONTIER(T_0)

$target \leftarrow \max\{1, \texttt{workers} \times m(C^*)\}$ *frontier-multiplier ladder; $m(C^*)$ from the table in §3*

`frontier` $\leftarrow \{T_0\}$; `best_gain_map`[key(T_0)] $\leftarrow T_0$.`current_gain` *seed dominance map with the root state*

while |`frontier`| < *target* **do** *expand until target size reached or search terminates*

- *if* `abort` *then* `frontier` $\leftarrow \varnothing$; **break** *gap closed or time limit reached; abandon frontier*
- $T \leftarrow$ task in `frontier` with maximum `active_count`(T) *ties broken by smaller* `current_gain`
- remove T from `frontier`
- APPLYEXCLUSIVEDEPENDENCYRULE(T) *full metadata rebuild; may increase T.`current_gain`*
- *if* key(T) ∈ `best_gain_map` *with stored gain* > T.`current_gain` *then* **continue** *strictly dominated: a sibling reached the same state with strictly greater gain*
- $ub \leftarrow$ LagrangianBound(T, min(64, 8 + ⌊`active_count`/8⌋)) *projected-subgradient gain upper bound; also tightens root lb*
- *if* `abort` *then* `frontier` $\leftarrow \varnothing$; **break** *gap may have closed while computing ub*
- *if* ub *cannot beat incumbent then*
  - *if* `frontier` = ∅ *then* **break**; **else continue** *T pruned; keep expanding if other tasks remain*
- (b, ub_{exc}) $\leftarrow$ ChooseBranchLine(T, ub) *gain/slack/pressure shortlist; exclude bound precomputed per candidate*
- *if* $b = \varnothing$ *then* **submit** T; *if* `frontier` = ∅ *then* **break**; **else continue** *no productive line: T is a complete candidate; break if no tasks remain*
- $ub_{\text{inc}} \leftarrow$ IncludeUpperBound(T, b, ub) *Lagrangian bound on the include child of b*
- `children` $\leftarrow \langle\,\rangle$ *at most two children (include and exclude)*
- *if* ub_{inc} *can beat incumbent then* *include branch*
  - $T_{\text{inc}} \leftarrow$ copy(T)
  - $\delta \leftarrow \delta_T(b)$ *= $c_T(b) - 2$; positive gain from selecting b*
  - *if* $\delta > 0$ *then*
    - T_{inc}.`current_gain` += δ; T_{inc}.`current_choice`.push(b)
    - SelectLineWithoutUndo(T_{inc}, b) *update 1024-bit mask and residual cover counts; no undo log*
    - append T_{inc} to `children`
- *if* ub_{exc} *can beat incumbent then* *exclude branch*
  - $T_{\text{exc}} \leftarrow$ move(T) *T consumed: exclude is always the last branch*
  - DisableLineWithoutUndo(T_{exc}, b) *mark b unavailable; update productive and blocked-productive sets*
  - append T_{exc} to `children`
- **for each** $C \in$ `children` **do** *incremental forcing, deduplication, and frontier insertion*
  - ApplyIncrementalForcing(C) *EDR via SELECTLINEWITHOUTUNDO; no full metadata rebuild*
  - *if* key(C) *not dominated in* `best_gain_map` *then* append C to `frontier` *push only if novel or strictly better; record state either way*

sort `frontier`: descending `active_count`, then descending |`productive_ids`|, then ascending `current_gain` *harder, less-pruned tasks dispatched first*

return `frontier`

Notation: key(T) := $\big(U(T), B(T)\big)$ where $U(T)$ is the 1024-bit active-point mask and $B(T)$ the sorted blocked-productive id list (theorem 2.8); $\delta_T(b) := c_T(b) - 2$ (`positive_gain`); C^* is the current best-known cover cost; dominance check and map update are fused in `remember_frontier_state`(T, `best_gain_map`) (discards when stored gain ≥ child's gain, updates otherwise); `frontier_state_is_stale` performs a read-only check for the extracted task T (discards only when stored gain > T.`current_gain`, so equal-gain siblings are not suppressed).

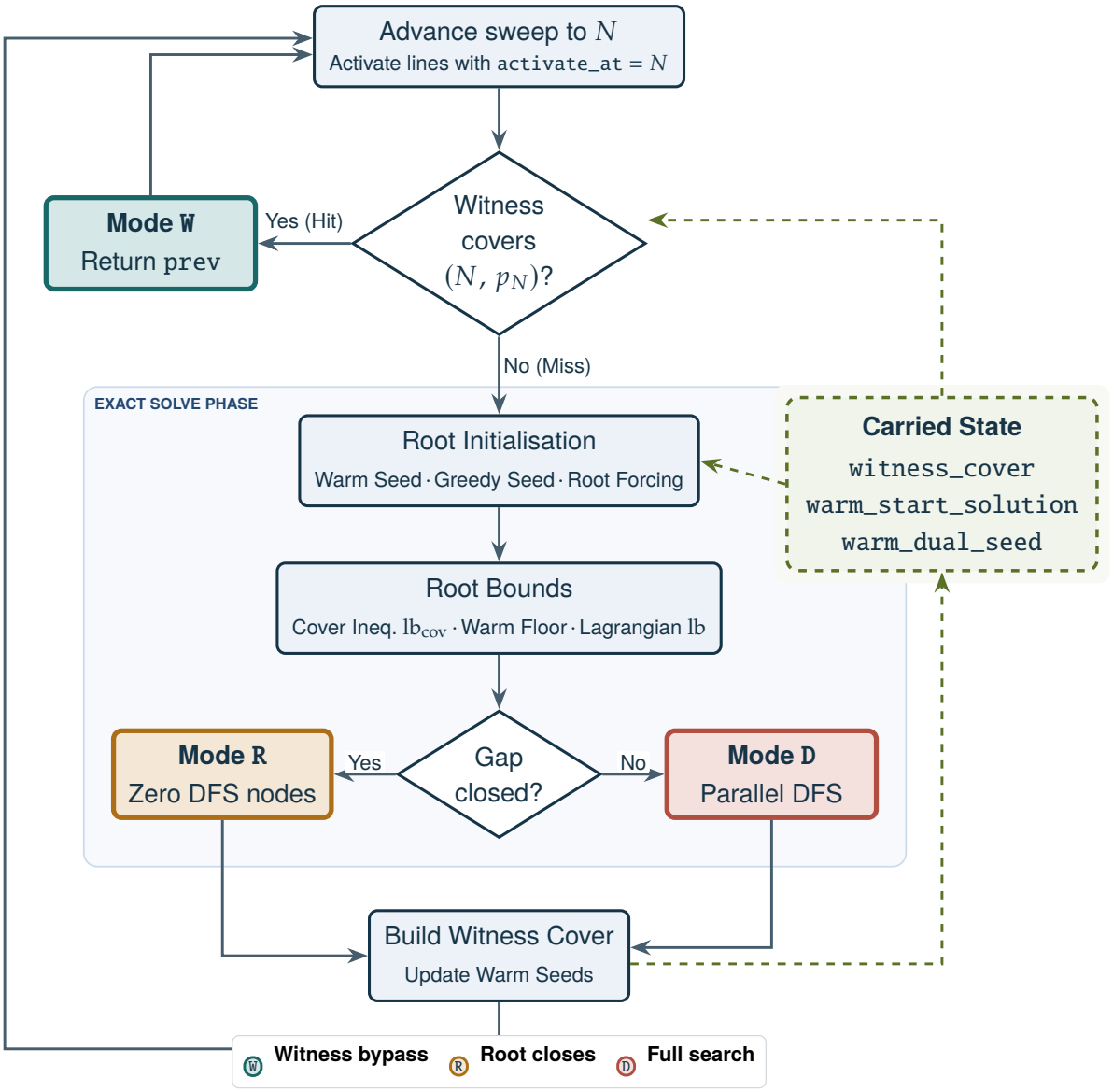

Figure 5: Algorithm architecture and carried-state propagation across the incremental sweep. At each step N the driver appends to `active_ids` every heavy line whose `activate_at` equals N, then evaluates the carried `witness_cover`. A hit (**Mode W**) bypasses the exact solver entirely and returns `prev` immediately. On a miss the driver calls `solve_exact()`, which constitutes the *exact solve phase*: it initialises the root node from the carried `warm_start_solution` and `warm_dual_seed`, computes root bounds from the cover inequality, the previous step's certified cost, and the Lagrangian bound, and then either closes the gap without DFS (**Mode R**, zero nodes counted) or launches the parallel frontier and DFS (**Mode D**). Both exact modes conclude by rebuilding the witness cover and writing back updated warm seeds for the next step. Dashed arrows show the three carried-state channels.

4 Incremental Warm-Start and Witness Propagation

4.1 Carried State and the Three Modes

The incremental structure is the central design feature of the driver. Between successive values of N the driver carries five items:

- `prev`, the most recently certified cover size $f(N-1)$;
- `active_ids`, the set of *active heavy lines*—heavy lines whose `activate_at` value is at most the current N, equivalently those containing at least three points of $\mathcal{S}_N$;
- `warm_start_solution`, the heavy-line set written back into this variable by `solve_exact` at the end of every call, timed-out or not;
- `warm_dual_seed`, the Lagrangian dual vector exported by the most recent call to `solve_exact` that produced a non-empty dual seed (in practice, every non-timed-out call); and
- `witness_cover` together with the Boolean flag `witness_ready`.

A heavy line is assigned `activate_at` $= p_3 + 1$, where p_3 is the zero-indexed position of its third point; the condition `activate_at` $\le N$ is therefore equivalent to requiring that the third point lies within $\mathcal{S}_N$. Points are represented in one-indexed coordinates throughout: the k-th point is $(k,\ p_k)$, where p_k is the k-th prime.

The driver distinguishes three execution modes for each N.

Mode `W` (witness hit). The driver enters mode `W` if and only if `witness_ready` is `true` and the call `witness_covers_point(witness_cover,` N, p_N`)` returns `true`. In that case the driver sets `exact_result.answer`, `exact_result.lb`, and `exact_result.lb_raw` all to `prev`, and does not call `solve_exact`. The carried witness, warm heavy-line set, and warm dual seed therefore remain unchanged.

Mode `R` (exact solve, zero DFS nodes). If the witness misses, the driver calls `solve_exact`. Mode `R` is recorded when the field `nodes` in the returned `ExactSolveResult` equals zero. This field accumulates one count per DFS node at the `++nodes` statement inside `dfs()`, so it stays zero whenever the root interval—defined by `current_best_cost()` $\le$ `root_cost_lb_` and gated on the Boolean `root_lb_captured_`—closes before any worker reaches that statement. On the multi-worker path this occurs if the root interval closes during `build_frontier` (which evaluates the Lagrangian bound and thereby sets `root_lb_captured_`), or if every frontier task is pruned at one of the three guards preceding `++nodes` in `run_task` and `dfs`. Algorithmically, mode `R` is not a distinct solver: it is the case where seeding, forcing, greedy completion, and bounding already certify the incumbent before branch-and-bound counts a single node.

Mode `D` (exact solve, positive DFS count). If the witness misses and the returned `nodes` field is positive, the driver records mode `D`.

In the full sweep over all $N = 1, \ldots, 1024$, the run contains 615 witness hits, 236 mode-`R` rows, and 173 mode-`D` rows.

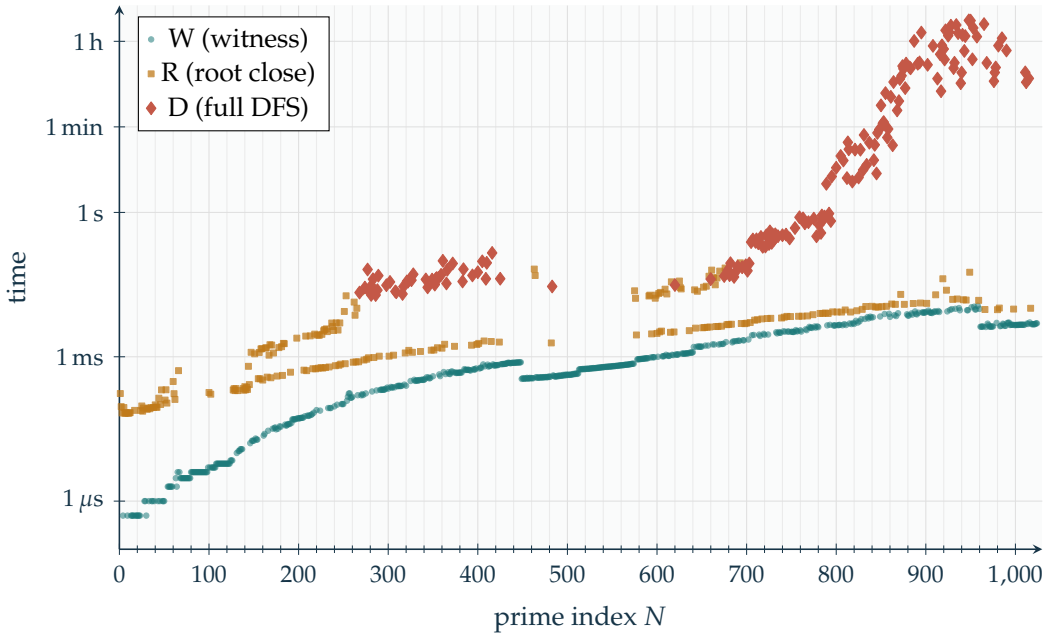

Figure 6: Per-instance solve time on a logarithmic scale across all 1024 certified indices, distinguished by execution mode. Circles (`W`) are witness hits; squares (`R`) are root closures requiring no DFS nodes; diamonds (`D`) are full depth-first searches. The seven-order-of-magnitude spread—from sub-microsecond witness hits to multi-hour DFS runs—makes the witness mechanism's value immediately visible. Across all 1024 indices the mode split is 615 witness hits (`W`, 1.1 s total), 236 root closures (`R`, 1.9 s total), and 173 full DFS runs (`D`, 142,961.9 s total); the 173 DFS instances therefore account for 99.998% of total compute time despite representing only 16.9% of all indices.

4.2 Witness Construction

After every successful (non-timed-out) call to `solve_exact`, the driver constructs a witness cover by calling

$$\texttt{build\_witness\_cover}(N,\ N_{\text{end}},\ \texttt{primes},\ \texttt{lines},\ \texttt{warm\_start\_solution}),$$

and sets `witness_ready = true`. Because `solve_exact` writes the chosen heavy-line set back into `warm_start_solution` unconditionally before returning (see section 4.3), the witness is always built from the optimal solution found for $\mathcal{S}_N$. The horizon argument N_{end} is the final index of the requested sweep, enabling the future-aware pairing described below.

Each line is stored as a `LineKey` (a, b, c) satisfying $ax+by+c = 0$ in one-indexed coordinates, reduced to lowest terms with positive leading nonzero coefficient. `witness_covers_point(witness,` N, p_N) returns `true` if and only if some stored line passes through (N, p_N).

The construction is deterministic and proceeds as follows.

1. For each heavy line in the chosen set, the routine computes the canonical line through the one-indexed coordinates of its first two points and appends the corresponding `LineKey` to the witness. Every point of the heavy line that lies in $\mathcal{S}_N$ is marked covered.
2. The uncovered (residual) zero-indexed positions of $\mathcal{S}_N$ are collected in increasing order into a list R.
3. If $N < N_{\text{end}}$ and $|R| \geq 2$, the routine enters the *future-aware pairing phase*. It scans future indices $t = N + 1, N + 2, \ldots$ in order until it finds the first index not yet covered by the witness assembled so far (including coverage by the heavy-line keys from step 1).

4. Among all unused residual pairs $(i, j) \subseteq R$ whose canonical line through $(i+1,\ p_{i+1})$ and $(j+1,\ p_{j+1})$ passes through $(t,\ p_t)$, the routine selects the pair maximising:

   (a) the *reach*: the length of the maximal prefix of $t,\ t+1,\ \ldots$ in which every index is either already covered or lies on the candidate line;

   (b) among ties, the total count of not-yet-covered future indices from t through N_{end} that the candidate line covers; and

   (c) among remaining ties, the lexicographically smallest pair (i, j).

   The selected pair is removed from R, its canonical line is appended to the witness, and the future-covered array is updated.

5. Steps 3–4 repeat until every future index through N_{end} is covered or no unused residual pair passes through the next uncovered future index.

6. The remaining elements of R are paired in increasing order, each pair contributing one line through its two one-indexed points. If an odd element $p \in R$ is left over, the routine appends the vertical line $x = p + 1$ as the `LineKey` $(1,\ 0,\ -(p+1))$.

Proposition 4.1. *Let* `chosen_line_ids` *be any heavy-line set for* $\mathcal{S}_N$*. Then* `build_witness_cover` *returns a cover of* $\mathcal{S}_N$ *with exactly*

$$|\texttt{chosen\_line\_ids}| + \left\lceil \frac{r}{2} \right\rceil$$

lines, where $r = |R|$ *is the number of points of* $\mathcal{S}_N$ *not covered by the chosen heavy lines.*

Proof. Each chosen heavy line contributes one geometric line and covers precisely the points marked in step 1. Every line appended during the future-aware phase (step 4, across all repetitions) and during the final sequential pairing (step 6) uses either two residual points or one residual point plus a vertical singleton; no residual point appears in more than one such line. Thus all r residual points are covered by exactly $\lceil r/2 \rceil$ additional lines. □

When `warm_start_solution` is optimal for $\mathcal{S}_N$, theorem 4.1 and theorem 2.6 together imply that the witness has exactly $f(N)$ lines.

4.3 Warm Heavy-Line Seeds and Warm Dual Seeds

The warm start has two independent components.

Warm heavy-line set. `solve_exact` writes the chosen heavy-line set into `warm_start_solution` unconditionally before returning, whether or not the solve timed out. (A timeout also terminates the outer sweep immediately, so the updated value is never reused in practice.) On the next witness miss, `solve_for_n` generates incumbent candidates from four passes before branch-and-bound begins.

1. `seed_from_line_ids` is called with `warm_start_solution`. It copies the unmodified root task into a trial, replays each line in order—skipping any whose *positive gain* $\max(0,\ \texttt{line\_cover}[id] - 2)$ on the trial state is zero, and skipping unavailable lines (whose positive gain is defined to be zero)—and submits the resulting partial solution as a candidate if its total gain strictly exceeds the current best gain. The root task is not modified.

2. `seed_from_line_ids` is called again with the greedy upper-bound line set (`greedy_seed_line_ids`) via the same trial-copy mechanism.

3. `apply_root_forced_line_reduction` modifies the root task in place by forcing lines according to the Exclusive Dependency Rule. `greedy_completion` is then called on the forced root; if the resulting greedy gain strictly exceeds the current best gain, the greedy line set is submitted as a candidate.

4. `greedy_complete_from_seed` is called with `warm_start_solution`. Starting from a copy of the force-reduced root (step 3's output), it applies each warm-start line whose positive gain is nonzero, then extends the partial solution by greedy completion and submits if the combined gain strictly exceeds the current best gain.

These are the only places in the search where the carried combinatorial solution is used directly.

`submit_candidate` updates `best_choice_` only on a strict gain improvement, so the first solution attaining the final optimal gain is stored and later equal-gain candidates do not replace it. After `solve_for_n` returns, `solve_exact` writes `chosen_line_ids` back into `warm_start_solution` unconditionally; only a non-timed-out solve then calls `build_witness_cover` and sets `witness_ready = true`.

Warm dual seed. The Lagrangian subproblem in `lagrangian_upper_bound` maintains a per-solver-instance dual vector `initial_dual_seed_` that is propagated across successive values of N. The local closure `store_best_dual` writes the current best dual values into this seed in three circumstances: at an early exit when the initial bound (evaluated before any subgradient step) already certifies pruning; inside `maybe_confirm_prune` when exact evaluation during a subgradient iteration confirms pruning; and unconditionally after the coordinate-descent polish that concludes every evaluation that does not exit early. The flag `root_dual_seed_captured_` ensures that only the *first* invocation of `store_best_dual` within a given call to `solve_for_n` writes to `initial_dual_seed_`; subsequent invocations update the per-worker Lagrangian scratch buffer `best_y` but leave the exported seed frozen.

This frozen seed is reused in two ways. First, `prepare_worker_context` loads `initial_dual_seed_` into the scratch buffer `best_y` of every freshly prepared worker context, so that all bound computations—in `run_task`, `build_frontier`, or a parallel worker thread—start from the same warm dual point rather than from zero. Second, `solve_for_n` returns the frozen seed as `out.dual_seed`, and `solve_exact` propagates it into `warm_dual_seed` (provided `out.dual_seed` is non-empty) before returning, making it available as the incoming seed for the next value of N.

One distinction is worth noting: the witness and the warm heavy-line set are both derived from the last successful complete solve taken as a whole, whereas the exported dual seed captures the dual state at the *first* invocation of `store_best_dual` within that solve.

5 Lower Bound

5.1 The Root Cover Inequality

Let $\mathcal{S}_N$ denote the point set $\{(i, p_i)\}_{i=1}^{N}$, and let $[N] = \{1, \ldots, N\}$ index those points. A line is *heavy* if it contains at least three points of $\mathcal{S}_N$, and *active at* N if its third point (ordered by index) has index at most N; the set of all such lines is A_N.

Definition 5.1. *A set $S \subseteq [N]$ is* 2-cover-free *if every line in A_N contains at most two points of S.*

Proposition 5.2. *If $S \subseteq [N]$ is 2-cover-free, then every line cover of $\mathcal{S}_N$ uses at least $\lceil |S|/2 \rceil$ lines.*

Proof. Any non-heavy line contains at most two points of $\mathcal{S}_N$, hence at most two points of S. By the 2-cover-free property every active heavy line also contains at most two points of S.

Therefore every line in any cover accounts for at most two points of S, so at least $\lceil |S|/2 \rceil$ lines are necessary. □

The function `cover_inequality_lb()` is called once per non-witness solve, before the branch-and-bound engine starts. It does not solve the maximum-cardinality 2-cover-free problem exactly. Instead it starts from $S = [N]$, computes for each point $p \in S$ its *violation count*—the number of lines in A_N currently covering at least three points of S that contain p—and repeatedly removes the point with the largest violation count until no violated line remains, then returns $\lceil |S|/2 \rceil$.

The result is stored as `lb_cov`. Together with the monotonicity floor

$$\texttt{lb0\_floor} = \begin{cases} \texttt{prev} & \text{if a warm-start solution from } N-1 \text{ is available,} \\ 0 & \text{otherwise (first } N\text{),} \end{cases}$$

where `prev` is the exact minimum cover cost for $N-1$, this yields the effective floor

$$\texttt{lb0\_floor\_eff} = \max\{\texttt{lb0\_floor},\ \texttt{lb\_cov}\},$$

which is passed to the branch-and-bound engine as `root_cost_lb_floor_` before any Lagrangian bound is computed.

5.2 Lagrangian Cost Lower Bound

At a search node T, the *residual coverage* of a heavy line h is

$$c_T(h) := |\mathrm{pts}(h) \cap U(T)|,$$

where $U(T)$ is the set of active (uncovered) points. A line is *productive* at T if it is available and $c_T(h) \geq 3$; the set of productive lines is $P(T)$. Define

$$r_T(h) := c_T(h) - 2, \qquad A_T(h) := \mathrm{pts}(h) \cap U(T), \qquad U^+(T) := \bigcup_{h \in P(T)} A_T(h).$$

Dual variables are materialized only for points in $U^+(T)$. For $y = (y_i)_{i \in U^+(T)}$ with $0 \leq y_i \leq 1$, define

$$L_T(y) := \sum_{i \in U^+(T)} y_i + \sum_{h \in P(T)} \max\bigl(0,\ r_T(h) - \textstyle\sum_{i \in A_T(h)} y_i\bigr). \tag{1}$$

Proposition 5.3. *For every node T and every feasible y, the quantity $\texttt{current\_gain}(T) + L_T(y)$ is an upper bound on the total gain attainable at or below T.*

Proof. Fix any continuation below T and list the selected productive lines $h_1, \ldots, h_m$ in selection order. Let $C_t \subseteq A_T(h_t)$ be the points first covered by h_t, so the gain contribution of h_t is $|C_t| - 2$, and writing $O_t = A_T(h_t) \setminus C_t$ we have $|C_t| - 2 = r_T(h_t) - |O_t|$.

If $r_T(h_t) - \sum_{i \in A_T(h_t)} y_i > 0$ then, since each $y_i \leq 1$,

$$\max\bigl(0, r_T(h_t) - \textstyle\sum_{i \in A_T(h_t)} y_i\bigr) + \displaystyle\sum_{i \in C_t} y_i = r_T(h_t) - \sum_{i \in O_t} y_i \geq r_T(h_t) - |O_t| = |C_t| - 2.$$

If instead $r_T(h_t) - \sum y_i \leq 0$ then $\sum_{i \in C_t} y_i = \sum_{A_T(h_t)} y_i - \sum_{O_t} y_i \geq r_T(h_t) - |O_t| = |C_t| - 2$. In both cases

$$|C_t| - 2 \leq \max\bigl(0, r_T(h_t) - \textstyle\sum_{i \in A_T(h_t)} y_i\bigr) + \displaystyle\sum_{i \in C_t} y_i.$$

Summing over t and using disjointness of the C_t gives total additional gain at most $L_T(y)$, completing the proof. □

Root cost lower bound. After each Lagrangian evaluation inside the branch-and-bound engine, the solver calls `capture_root_cost_lb()`; the flag `root_lb_captured_` ensures that only the first such call records its result. Writing Γ for the total gain bound returned by that call, the *cost-from-gain* map $\mathrm{cost}(N, G) = \lceil (N - G)/2 \rceil$ gives

$$\mathrm{lb}_{\mathrm{raw}} = \mathrm{cost}\big(N,\ \lfloor \Gamma + 10^{-9} \rfloor\big) = \left\lceil \frac{N - \lfloor \Gamma + 10^{-9} \rfloor}{2} \right\rceil.$$

The tolerance 10^{-9} prevents floating-point rounding from deflating an integrally exact Γ. The final root lower bound is

$$\mathrm{lb} = \max\{\texttt{lb0\_floor\_eff},\ \mathrm{lb}_{\mathrm{raw}}\} = \max\{\texttt{lb0\_floor},\ \mathrm{lb}_{\mathrm{cov}},\ \mathrm{lb}_{\mathrm{raw}}\}.$$

5.3 Subgradient Descent and Coordinate Polish

The implementation minimizes (1) by projected subgradient descent. For fixed y and $h \in P(T)$, define the *line slack*

$$\sigma_h(y) \coloneqq r_T(h) - \sum_{i \in A_T(h)} y_i.$$

The subgradient at point $i \in U^+(T)$ is

$$g_i = 1 - \#\{h \in P(T) : \sigma_h(y) > 0,\ i \in A_T(h)\}.$$

Let C denote the current incumbent cost and let `current_choice` denote the ordered sequence of heavy-line IDs selected so far in the current DFS path. The minimum gain required to strictly improve the incumbent is $G_{\mathrm{need}} = N - 2C + 2$, and the *remaining gain target* at node T is

$$\tau_T = \max\big(0,\ G_{\mathrm{need}} - \texttt{current\_gain}(T)\big).$$

With $y^{(k)}$ the iterate at step k, the projected update is $y_i^{(k+1)} = \mathrm{clip}_{[0,1]}\big(y_i^{(k)} - \alpha^{(k)} g_i^{(k)}\big)$, where the step size is

$$\alpha^{(k)} = \frac{\sigma_{\mathrm{step}}\ \max\big(0,\ L_T(y^{(k)}) - \tau_T\big)}{\|g^{(k)}\|_2^2}.$$

The *step scale* σ_{step} depends on the node's dual warm-start status and the current search state:

$$\sigma_{\mathrm{step}} = \begin{cases} 1.75, & \text{if warm-started, } C \geq 113, \text{ and } |\texttt{current\_choice}| \geq 4, \\ 1.5, & \text{if warm-started but the above condition fails,} \\ 1.35, & \text{otherwise.} \end{cases}$$

Here *warm-started* means `scratch.dual_ready` is `true` at the start of the evaluation. Each update is projected onto $[0, 1]^{|U^+(T)|}$.

Iteration budget. The budget is exactly

$$K(T) = \min\left(64,\ 8 + \left\lfloor \frac{|U(T)|}{8} \right\rfloor\right).$$

Staged warm-start. When the node is warm-started, $C \geq 112$, and $|\texttt{current\_choice}| \geq 4$, the solver first runs only $\min(K(T), 16)$ iterations. Let L_T^* denote the best (minimum) Lagrangian value achieved during that first stage. The solver then proceeds to the full budget $K(T)$ only if

$$\texttt{current\_gain}(T) + L_T^* < G_{\mathrm{need}} + 1, \tag{2}$$

i.e., the best total gain bound from the first stage falls within 1 of the pruning threshold G_{need} (equivalently, $L_T^* < \tau_T + 1$).

Initialization. If a dual seed is available (`scratch.dual_ready` is `true`), each coordinate of $y^{(0)}$ is set to the corresponding stored entry in `best_y`, clamped to $[0, 1]$. Otherwise, for each productive line $h \in P(T)$ with coverage $k = c_T(h)$, the density $(k-2)/k$ is computed, and each active point of h receives the maximum density over all productive lines through it, clamped to 1.

Coordinate polish. After subgradient descent, the solver runs a single coordinate-polish sweep when the best total gain bound lies within 0.5 of G_{need}, i.e.,

$$\texttt{current\_gain}(T) + L_T^* < G_{\text{need}} + 0.5.$$

For a single coordinate y_i with all other coordinates fixed, the objective (1) has the form $y_i + \sum_{h \ni i} \max(0, a_h - y_i)$, where $a_h := \sigma_h(y)\big|_{y_i=0}$ is the adjusted line slack of h at $y_i = 0$. The objective is piecewise linear with breakpoints at the positive values of a_h, so the minimizer over $[0, 1]$ is the second-largest positive adjusted slack clamped to $[0, 1]$; this is the value assigned in the source.

5.4 Branch-Specific Bound Adjustments

After choosing a branch line b, the solver computes tighter gain caps for each child without re-running the Lagrangian. Let

$$\Gamma := \texttt{current\_gain}(T) + L_T(y)$$

denote the parent total gain bound of Theorem 5.3.

Exclude child. The parent term $\max(0, \sigma_b(y))$ vanishes immediately. For each active point $p \in A_T(b)$, let

$$R_p(b) = \big\{h \in P(T) \setminus \{b\} : 0 < \sigma_h(y) \le 1 - y_p,\ p \in A_T(h)\big\}.$$

The exclude cap is

$$\text{cap}_{\text{excl}} = \Gamma - \max(0, \sigma_b(y)) - \sum_{p \in A_T(b)} \Big(\sum_{h \in R_p(b)} \sigma_h(y) - \max_{h \in R_p(b)} \sigma_h(y) \Big),$$

clipped below by $\texttt{current\_gain}(T)$.

Include child. Selecting b contributes the definite gain $\delta_T(b) = c_T(b) - 2$. For each productive competitor $h \neq b$, define

$$\rho_h(b) := \sum_{p \in A_T(b) \cap A_T(h)} (1 - y_p).$$

The include cap is

$$\text{cap}_{\text{incl}} = \Gamma + \min(0, \sigma_b(y)) + \sum_{h \in P(T) \setminus \{b\}} \big(\max(0, \sigma_h(y) - \rho_h(b)) - \max(0, \sigma_h(y))\big),$$

clipped below by $\texttt{current\_gain}(T) + \delta_T(b)$.

Proposition 5.4. *Both child caps are valid upper bounds on the total gain attainable in the respective child.*

Proof. I start from the valid parent bound of Theorem 5.3 and write

$$\Gamma = \texttt{current\_gain}(T) + \sum_{i \in U^+(T)} y_i + \sum_{h \in P(T)} \max(0, \sigma_h(y)).$$

Include child. Selecting b contributes the definite gain $\delta_T(b) = c_T(b) - 2$, removes every $p \in A_T(b)$ from the active set (losing $\sum_{p \in A_T(b)} y_p$ from the point term), and for each competitor $h \neq b$ reduces $r_T(h)$ by 1 for each $p \in A_T(b) \cap A_T(h)$; the corresponding y_p also leaves the point sum, so $\sigma_h(y)$ shifts to $\sigma_h(y) - \rho_h(b)$. The change from Γ is therefore

$$\underbrace{r_T(b) - \textstyle\sum_{p \in A_T(b)} y_p}_{= \sigma_b(y)} + \sum_{h \neq b} \big(\max(0, \sigma_h(y) - \rho_h(b)) - \max(0, \sigma_h(y))\big),$$

giving the stated include cap (before the floor, which only strengthens the bound).

Exclude child. Since two distinct geometric lines share at most one point, the sets $R_p(b)$ for distinct $p \in A_T(b)$ involve disjoint collections of competitors. Fix p and any feasible continuation in the exclude child. Among the lines of $R_p(b)$ selected by the continuation, at most one can cover p first. For every other selected line $h \in R_p(b)$, point p lies in O_h (already covered), so the excess of the parent bound over h's realized gain satisfies

$$\max(0, \sigma_h(y)) + \textstyle\sum_{i \in C_h} y_i - (|C_h| - 2) = \sum_{q \in O_h} (1 - y_q) \geq 1 - y_p \geq \sigma_h(y),$$

where the last inequality uses $h \in R_p(b)$. Hence the parent bound overestimates h's realizable contribution by at least its full slack $\sigma_h(y)$. Summing these excesses over all points $p \in A_T(b)$, and also removing the now-forbidden term $\max(0, \sigma_b(y))$, yields the stated exclude cap (before the floor). □

The full-sweep log beyond 861 shows that on the 74 non-witness rows in the new block the root cover bound satisfies $78 \leq \mathrm{lb}_{\mathrm{cov}} \leq 94$ and the root Lagrangian cost bound satisfies $112 \leq \mathrm{lb}_{\mathrm{raw}} \leq 131$. On those same 74 rows the root diagnostic gap `ub0 - lb` is always exactly 1. The carried floor `lb0_floor` ranges from 123 to 143 and dominates the final root lower bound throughout the block. On the 53 mode-`D` rows the log records $26{,}423{,}637 \leq$ `lag_iters` $\leq 12{,}591{,}626{,}629$, $498{,}757 \leq$ `lag_prune` $\leq 238{,}420{,}400$, strong-branching counts between 3 and 18, and maximum search depth 108.

6 Correctness and Optimality

We collect the terminology used throughout this section. A *heavy line* is any line in $\mathbb{R}^2$ that passes through at least three points of $\mathcal{S}_N$. At a search node T, a heavy line is *available* if it has not yet been excluded, and *productive* at T if it is available and its *residual coverage*—the number of active (uncovered) points of $\mathcal{S}_N$ it contains—is at least three. Write $U(T)$ for the active point set and $B(T)$ for the set of unavailable lines whose residual coverage is still at least three. The *accumulated gain* at T is

$$\mathrm{gain}(T) \;:=\; \sum_{\substack{h \text{ selected} \\ \text{above } T}} \big(\mathrm{cov}(h) - 2\big),$$

where $\mathrm{cov}(h)$ is the residual coverage of h at the moment of its selection. When T has no productive line, the $|U(T)|$ remaining active points are covered by $\lceil |U(T)|/2 \rceil$ lines (pairs plus at most one singleton), so the final cover size equals $\lceil (N - \mathrm{gain}(T))/2 \rceil$. This identity is stated precisely in theorem 2.6.

Lemma 6.1 (Monotonicity)**.** *For every $N \geq 1$, $f(N+1) \geq f(N)$.*

Proof. Any cover of $\mathcal{S}_{N+1}$ restricts to a cover of $\mathcal{S}_N$, because $\mathcal{S}_N \subseteq \mathcal{S}_{N+1}$. Therefore the optimum cannot decrease when a point is added. □

6.1 The Exclusive Dependency Rule

Definition 6.2. *For a productive line h at node T, define*

$$E_T(h) \;:=\; \{i \in U(T) : h \text{ is the unique productive line through } i\}.$$

Theorem 6.3 (Exclusive Dependency Rule)**.** *If $|E_T(h)| \geq 3$, then some optimal completion below T contains h.*

Proof. Fix an optimal completion below T that omits h. Every point of $E_T(h)$ must then be covered by a non-productive line. Any non-productive line has residual coverage at most two, so if it covers at least one point of $E_T(h)$ it covers at most one active point outside $E_T(h)$.

Let k be the number of non-productive lines covering at least one point of $E_T(h)$, and let $d \leq k$ be the number of active points outside $E_T(h)$ that those k lines cover. Replace those k lines with h (cost 1), then cover the d displaced outside points with at most $\lceil d/2 \rceil \leq \lceil k/2 \rceil$ pair lines. The total cost change is

$$\Delta(k) \;:=\; -k + 1 + \left\lceil \tfrac{k}{2} \right\rceil.$$

For $k = 2$, $\Delta(2) = 0$; for $k = 3$, $\Delta(3) = 0$; and for $k \geq 4$, $\Delta(k) \leq -k/2 + 3/2 < 0$. Since each non-productive line covers at most two points of $E_T(h)$, covering $|E_T(h)| \geq 3$ of them requires at least $\lceil 3/2 \rceil = 2$ such lines, so $k \geq 2$. Hence $\Delta(k) \leq 0$ for all attainable k, and the replacement yields an optimal completion that contains h. □

The solver applies this rule iteratively: at each pass it selects the productive line with the largest $|E_T(\cdot)| \geq 3$ (ties broken first by residual gain, then by smallest line index) and forces it into the partial solution, repeating until no productive line satisfies the threshold. Each forced selection is safe by theorem 6.3 applied to the updated node.

6.2 Frontier Dominance

Lemma 6.4 (Frontier dominance)**.** *Suppose two nodes T_1 and T_2 satisfy*

$$U(T_1) = U(T_2), \qquad B(T_1) = B(T_2), \qquad \text{gain}(T_1) \geq \text{gain}(T_2).$$

Then discarding T_2 cannot remove an optimal solution.

Proof. The active set $U(T)$ determines the residual coverage of every line. A line with residual coverage at most two can never regain productivity because U only shrinks; consequently $B(T)$, which tracks exactly the excluded lines whose residual coverage is still at least three, captures every future branching option. Hence T_1 and T_2 have identical future productive options and identical future gain opportunities. Every continuation available below T_2 is also available below T_1 with at least as much accumulated gain, so T_1 dominates T_2. □

6.3 Exact Solve and the Three Modes

Theorem 6.5. *Every call to `solve_exact()` returns an optimal answer.*

Proof. The solver searches over include/exclude decisions for productive heavy lines. By theorem 2.6, once no productive line remains, $\text{gain}(T)$ already determines the optimal final cover size $\lceil (N - \text{gain}(T))/2 \rceil$, so no further branching on the residual active points is needed.

Before the main branching search, `solve_exact` establishes a lower-bound floor from two independent sources: the cover-inequality bound $\lceil |S|/2 \rceil$, where $S \subseteq \{1, \ldots, N\}$ is a feasible subset produced by a greedy vertex-removal heuristic (which iteratively removes the point belonging to the greatest number of violated heavy lines—those covering three or more

points of the current S—until no heavy line covers more than two points of S), and the monotonicity lower bound $f(N-1)$, which is valid whenever a warm-start solution from the previous step is available. The cover-inequality bound is valid because any line—heavy, pair, or singleton—covers at most two points of S by construction, so any feasible cover requires at least $\lceil |S|/2 \rceil$ lines. The effective floor passed to the gain solver is the maximum of both bounds. At the root of the search (before any child branch is explored), the Lagrangian bound (theorem 5.3) is evaluated and incorporated: the final root lower bound is the maximum of the Lagrangian-derived value and the pre-computed floor, as stated precisely in section 5.2.

Every transformation the solver applies is safe. The Exclusive Dependency Rule is safe by theorem 6.3. Frontier-state deletion is safe by theorem 6.4. Root and node pruning by the Lagrangian bound are safe by theorem 5.3. The include and exclude child bounds are safe by theorem 5.4. The frontier builder reorganizes the same search tree into independent parallel tasks without discarding any task except by the rules already proved safe.

The solver exhausts every branch whose accumulated-gain bound can still improve on the incumbent, so the incumbent is globally optimal. □

Proposition 6.6 (Mode `W`)**.** *If the driver reports mode* `W` *at* N*, then the reported answer is* $f(N)$*.*

Proof. The current witness was built after an earlier successful exact solve at some step N' (the most recent step that required an exact solve). By theorems 2.6 and 4.1, the witness has exactly $f(N') =$ `prev` lines and covers all of $\mathcal{S}_{N'}$. Because no exact solve occurred between N' and N, the witness has never been rebuilt; thus every step from $N'+1$ to $N-1$ also fired mode `W` with `witness_covers_point` returning true, so the witness covers each of those intermediate points as well. In mode `W` the witness covers (N, p_N), hence it covers all of $\mathcal{S}_N$ in `prev` lines, giving $f(N) \le$ `prev`. Theorem 6.1 gives $f(N) \ge f(N-1) =$ `prev`, so $f(N) =$ `prev`. □

Proposition 6.7 (Mode `R`)**.** *If the driver reports mode* `R` *at* N*, then the reported answer is* $f(N)$*.*

Proof. Mode `R` means the witness missed, `solve_exact()` was called and returned, and the DFS node counter is zero. The counter is incremented inside each DFS call after forced-line reduction and a subsequent `root_interval_closed()` check, but before the Lagrangian pruning step; a zero count therefore indicates that every frontier task exited before reaching the counter.

Mode `R` arises exclusively in the multi-worker path. In the single-worker path `root_lb_captured_` is false when the first DFS call begins, so `root_interval_closed()` returns false at every guard in that call until after the first Lagrangian evaluation; consequently the counter is always incremented on the first call, and nodes ≥ 1. In the multi-worker path, `build_frontier` evaluates the Lagrangian on the root task and calls `capture_root_cost_lb`, which sets `root_lb_captured_` = true and records

$$\texttt{root\_cost\_lb\_} \;=\; \max\bigl(\text{raw Lagrangian LB},\ \text{floor}\bigr).$$

If this value meets or exceeds the incumbent, `root_interval_closed()` returns true, and every subsequent frontier task exits at the pre-dispatch check in `run_task`—or at the first guard inside `dfs`—without incrementing the counter. The same closure can arise when the pre-computed floor—the maximum of the cover-inequality bound and the monotonicity bound $f(N-1)$—already meets or exceeds the incumbent, so that `root_cost_lb_` does so as well as soon as `build_frontier` performs its first Lagrangian evaluation. By theorem 6.5, the returned answer is optimal. □

Proposition 6.8 (Mode `D`)**.** *If the driver reports mode* `D` *at* N*, then the reported answer is* $f(N)$*.*

Proof. Mode `D` means the witness missed, `solve_exact()` was called and returned, and the DFS node counter is positive. Theorem 6.5 gives the conclusion directly. □

Remark 6.9. The logged fields `lb`, `lb_raw`, `gap`, and `gap_raw` are root diagnostics, not final global gaps. A positive root gap in a non-witness row does not indicate an uncertified result: the exact search closes any remaining global gap to zero before termination. This is illustrated concretely by the 163 new certified values $N = 862, \ldots, 1024$ produced by `primecover1024.cpp` on a Google Cloud `c4d-highcpu-8` (15 GB RAM)—extending the prior record of $N = 861$ (certified by an industrial MIP solver in approximately 282 hours) in under 40 hours. Among these, the 20 new *awkward primes* at $N = 864, \ldots, 1015$, where cover sizes range from 124 to 143 lines, are the hardest: their non-witness rows carry a positive root gap of 1, yet in every case the exact search closes the remaining global gap to zero and certifies the answer before termination.

Corollary 6.10 (Certification policy)**.** *Every answer line printed by the program is certified optimal.*

Proof. Modes `W`, `R`, and `D` are exhaustive, and theorems 6.6 to 6.8 prove optimality in each case. □

7 OEIS Sequence and Data

OEIS entry A373813 [3] records, for each $N \geq 1$, the minimum number of lines needed to cover the prime-point set $\mathcal{S}_N = \{(i, p_i) : 1 \leq i \leq N\}$, where p_i denotes the i-th prime; we write $f(N)$ for this minimum. Figure 7 plots the certified staircase $N \mapsto f(N)$ over $N = 1, \ldots, 1024$.

This work certifies 163 new terms ($N = 862$ through $N = 1024$), raising the record from $f(861) = 123$—previously certified by an industrial mixed-integer programming (MIP) solver in approximately 282 hours on dedicated hardware [2]—to the new record $f(1024) = 143$. All 163 new values were certified by `primecover1024.cpp` on a single Google Cloud `c4d-highcpu-8` instance (15 GB RAM). The complete solver run from $N = 1$ to $N = 1024$ required 142,965 seconds (under 40 hours): the prior 861 certified values were reproduced in approximately 22 minutes, after which the 163 new certifications consumed the remaining ≈39.3 hours—roughly 104× the compute for the prior block.

An index N with $f(N) > f(N-1)$ is called an *awkward prime* (OEIS A393445 [4]; the prime p_N forces an additional cover line). Among the 163 new terms the cover size increases at exactly 20 indices; the corresponding *new awkward primes* p_{N^*} are:

$$6701,\ 6763,\ 6793,\ 6863,\ 6959,\ 6967,\ 7069,\ 7187,\ 7213,\ 7253,$$
$$7331,\ 7417,\ 7457,\ 7487,\ 7547,\ 7591,\ 7703,\ 7727,\ 7829,\ 8081.$$

Cover sizes in the new block span $f \in \{123, \ldots, 143\}$. Two terms ($N = 862, 863$) continue the prior plateau at $f = 123$; ten terms ($N = 1015, \ldots, 1024$) form the final plateau at $f = 143$; and the remaining 151 terms are distributed across the 19 intermediate plateaus. Table 2 summarises the 21 cover-size plateaus; table 3 gives solver diagnostics for each new awkward prime.

Solver modes. The solver assigns each index N one of three certification modes based on how optimality is established. A *heavy line* is a line passing through at least three prime points (i, p_i); a heavy line is *active* at step N once the 1-based index of its third prime point is at most N.

After each exact certification the solver constructs an explicit *witness cover*: a valid cover of size $f(N)$ in which the residual lines covering fewer than three prime points are chosen by a forward-looking greedy strategy—pairing uncovered residual points so as to extend coverage to as many future prime points as possible without adding a line.

A *witness row* (`W`) occurs when some line in the witness cover retained from step $N-1$ already passes through (N, p_N), certifying $f(N) = f(N-1)$ without invoking the exact solver; wall-clock time is under 15 milliseconds in every such case.

A *relaxation row* (`R`) occurs when the incumbent—seeded from the warm-start solution carried from step $N-1$ and refined by greedy completion over the active heavy lines—already meets the *root lower bound* after frontier expansion, so no branch-and-bound nodes need be explored; timing is likewise negligible (under 60 milliseconds). The root lower bound is the maximum of three components: the warm-start floor $f(N-1)$; the *cover-inequality bound* $\ell_{\text{cov}} = \lceil |S^*|/2 \rceil$, where $S^* \subseteq \{1, \ldots, N\}$ is the largest subset such that every active heavy line covers at most two of its points (since any line covers at most two elements of S^*, at least ℓ_{cov} lines are always required); and the Lagrangian relaxation bound, which is computed during frontier expansion by a subgradient method on per-point dual variables and provides an upper bound on the gain achievable by heavy-line selection.

A *DFS row* (`D`) requires branch-and-bound search and accounts for virtually all wall-clock time across the block; at each internal node the Lagrangian bound prunes subtrees whose gain ceiling falls below the current incumbent.

Across the 163 new terms, 89 are witness rows, 21 are relaxation rows, and 53 require DFS.

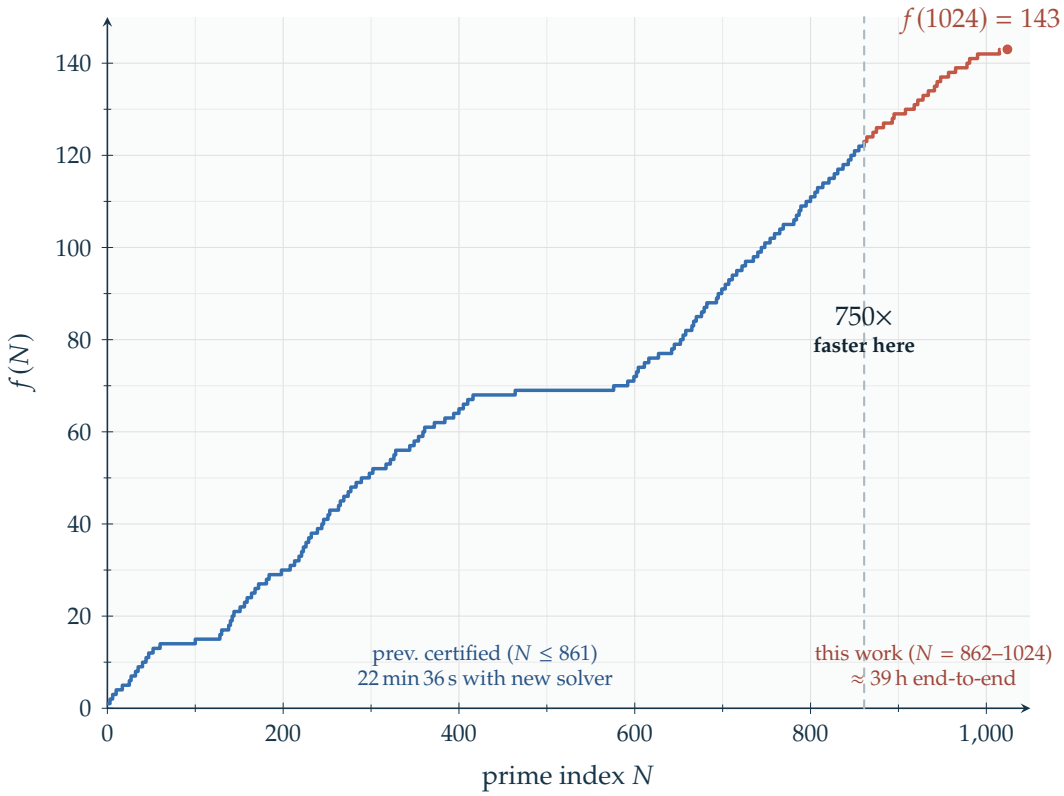

Figure 7: Minimum line cover number $f(N)$ over all certified values $N = 1, \ldots, 1024$. Each vertical step marks an *awkward prime*. Prior certified terms ($N \le 861$) are shown in a darker shade; the new solver reaches this boundary in 1355.9 s (22 min 36 s), versus 282 h 26 min 31.5 s for the prior MIP solver — a 749.9× speedup. The 163 new terms certified by `primecover1024.cpp` (N = 862–1024, ≈ 39 h end-to-end) extend the record from $f(861) = 123$ to $f(1024) = 143$. Sequence: OEIS A373813.

Table 2: Cover-size plateaus, N = 862–1024. N^*: awkward prime opening each plateau ("—" if $N^* \le 861$). D/R/W: DFS, relaxation, and witness row counts within the plateau. Wall-clock: summed per-row times from `A373813_STATISTICS.txt`; the $f = 123$ plateau covers N = 862–863 only. The grand total of 142,965 s includes ≈22 min for N = 1–861.

f	N^*	p_{N^*}	N-range	terms	D	R	W	wall-clock (s)
123	—	—	862–863	2	1	1	0	25
124	864	6701	864–870	7	3	1	3	851
125	871	6763	871–874	4	3	1	0	2 385
126	875	6793	875–882	8	4	1	3	3 620
127	883	6863	883–892	10	3	1	6	5 883
128	893	6959	893–894	2	1	1	0	1 310
129	895	6967	895–907	13	2	1	10	6 640
130	908	7069	908–917	10	4	1	5	5 767
131	918	7187	918–921	4	3	1	0	6 967
132	922	7213	922–927	6	3	1	2	16 454
133	928	7253	928–933	6	3	1	2	9 890
134	934	7331	934–940	7	4	1	2	14 477
135	941	7417	941–943	3	2	1	0	7 165
136	944	7457	944–947	4	1	1	2	4 632
137	948	7487	948–956	9	4	1	4	27 838
138	957	7547	957–964	8	1	1	6	4 499
139	965	7591	965–977	13	4	1	8	10 828
140	978	7703	978–980	3	1	1	1	1 034
141	981	7727	981–989	9	2	1	6	7 093
142	990	7829	990–1014	25	3	1	21	3 640
143	1015	8081	1015–1024	10	1	1	8	610
Total (N = 862–1024)			862–1024	163	53	21	89	141 608

Table 3: DFS certification details for each of the 20 new awkward primes. `active`: active heavy lines at index N (lines with ≥ 3 prime-point incidences whose third point has index $\leq N$). `nodes`: total branch-and-bound nodes explored. `lag_iters` / `lag_prune`: total Lagrangian subgradient iterations and nodes pruned by the Lagrangian bound. `strong_branch`: branch-and-bound nodes at which strong branching was invoked to select the branching variable. `depth`: maximum DFS depth reached. All values from `A373813_STATISTICS.txt`.

N^*	p_{N^*}	f	time (s)	active	nodes	lag_iters	lag_prune	strong_branch	depth
864	6701	124	512.711	9291	26 377 629	611 298 105	12 146 805	17	87
871	6763	125	592.927	9435	29 905 192	692 668 598	13 862 985	11	85
875	6793	126	965.594	9522	55 213 003	1 276 473 824	25 444 786	11	82
883	6863	127	988.772	9696	52 996 564	1 214 294 569	24 685 595	11	83
893	6959	128	1 309.949	9899	57 358 226	1 351 994 454	26 698 667	11	79
895	6967	129	5 473.864	9949	313 172 078	7 234 215 107	144 504 796	14	95
908	7069	130	2 893.717	10208	153 825 258	3 577 697 493	71 410 567	14	87
918	7187	131	2 962.099	10382	143 433 665	3 327 612 034	66 941 808	10	94
922	7213	132	6 091.884	10453	243 530 317	5 632 330 356	113 717 946	10	96
928	7253	133	7 595.066	10585	369 602 370	8 597 136 257	172 472 570	5	97
934	7331	134	7 848.330	10684	414 089 725	9 799 280 282	193 154 525	9	103
941	7417	135	4 891.252	10783	207 509 885	5 026 617 751	96 580 573	15	87
944	7457	136	4 632.034	10814	206 582 950	5 026 636 968	96 435 080	15	87
948	7487	137	9 797.923	10854	404 748 413	9 911 315 142	188 230 876	9	91
957	7547	138	4 499.133	10966	179 379 804	4 439 795 278	84 170 921	17	91
965	7591	139	8 207.569	11105	331 313 913	8 211 535 491	154 142 512	11	102
978	7703	140	1 034.366	11352	36 642 365	930 846 500	17 180 609	14	86
981	7727	141	2 925.120	11411	122 777 717	3 041 501 405	57 389 954	15	94
990	7829	142	2 317.121	11572	90 087 942	2 270 319 212	42 303 868	14	90
1015	8081	143	610.359	12012	14 491 781	383 323 440	6 759 783	14	93

8 Code Availability

The solver is a single C++23 source file, `primecover1024.cpp`. The suffix "1024" encodes two coincident constants in the `config` namespace: `kBitCapacity` = `1024`, the fixed width (in bits) of the coverage bitmasks used throughout the exact search, and `kExecutionLimit` = `1024`, the step at which the main sweep loop halts. A command-line argument `requested_n` is accepted but clamped to min(`kExecutionLimit`, `kBitCapacity`) = 1024, so the filename is a precise statement of the implementation's instance-size ceiling. The sweep begins at $N = 1$ (`kStartN` = `0` in the current source; the loop guard `std::max(kStartN, 1)` enforces this regardless of the constant's value).

The choice of $N = 1024$ as the design target was deliberate but uncertain. The prior certified boundary stood at $N = 861$, reached in 282 hours, and the rapid growth of the search space beyond $N = 800$ made extending the record to 1024 seem implausible at the outset. That the solver completes the full sweep in under 40 hours is a consequence of the architecture described in this paper, not an expectation that informed the choice of target.

Sweep log format. The sweep log `A373813_STATISTICS.txt` is the authoritative record for all 163 new terms. Each row corresponds to one value of N and carries a `mode` field identifying one of three execution paths. *Mode* `W` (*witness shortcut*): a forward-looking witness cover, constructed at the most recent mode-`D` or mode-`R` step and extended to cover prime points up to `requested_n`, already contains the new prime point (N, p_N), certifying $f(N) = f(N-1)$ without further search. *Mode* `R` (*root-only closure*): the root-level lower and upper bounds meet without branching (`nodes` = 0). *Mode* `D` (*direct branch-and-bound*): the parallel frontier search runs to completion with at least one branching node.

Two fields carry meaningful data for all three modes: `warm_size` (the number of heavy lines from the previous step's solution used to seed the warm start) and `greedy_heavy` (the number of heavy lines selected by the greedy upper-bound pass). Two further fields are valid for modes `R` and `D` and are zero for mode `W`: `prod`, the count of *productive* heavy lines at the root (those whose coverage count among the N active prime points is ≥ 3, the threshold at which a heavy line yields a net cost reduction over singleton coverage), and `pinc`, the sum of those coverage counts across all productive lines. The remaining solver-internal fields—`nodes`, `frontier`, `forced`, `forced_root`, `lag_iters`, `polish_sweeps`, `lag_prune`, `root_ub_frac`, `strong_branch`, and `depth`—are printed on every row but set to zero for modes `W` and `R`.

Frontier multiplier calibration. The parallel frontier capacity is `worker_count` $\times\, m(C)$, where $m(C)$ is a piecewise-constant multiplier indexed by the current line count C (the number of lines in the current best cover, i.e. $f(N)$ as the sweep progresses):

$$m(C) = \begin{cases} 8192 & C \geq 128, \\ 2048 & 125 \leq C < 128, \\ 512 & 121 \leq C < 125, \\ 128 & 113 \leq C < 121, \\ 16 & 93 \leq C < 113, \\ 4 & C < 93. \end{cases}$$

On the reference `c4d-highcpu-8` instance (8 vCPUs, 15 GB RAM) the top rung yields 8×8192 = 65,536 parallel tasks. The four lower rungs ($C < 93$ through $C < 125$) were fixed after extensive empirical testing across $N = 700$–825 over more than 1,000 solver iterations; the two upper rungs ($C \geq 125$ and $C \geq 128$) are estimates derived from the frontier diagnosis

tool and were not empirically tested. The multiplier governs peak RSS memory and parallel throughput, not correctness: any sufficiently large frontier produces a provably optimal answer (theorems 6.4 and 6.5); a frontier that is too small serialises work that could otherwise run in parallel, increasing wall-clock time without affecting the certified result. The solver was initially run with a fixed multiplier of 16384U; it crashed at $N = 924$, which prompted the development of `tools/primecover_frontier_diagnosis.sh`. The script compiles and runs the solver, collects RSS samples from $N = 808$ onward, fits the hyperbolic-decay model per-task$(F) = a + b/F$ to the observed RSS profile, and displays a live multiplier safety table.

Repository contents. The complete source code, precomputed results, and interactive demo are publicly available at:

`https://github.com/jespergran98/prime-line-cover`

An archived release is permanently deposited on Zenodo [6] at `https://doi.org/10.5281/zenodo.20096556` (DOI: `10.5281/zenodo.20096556`). The code is released under the MIT License; the data and results are dedicated to the public domain (CC0 1.0 Universal). The repository [5] contains:

- `primecover1024.cpp` and `primecover1024_line_coordinates.cpp`—the two solver variants described in this paper;
- `results/A373813_STATISTICS.txt`—the sweep log produced on the reference `c4d-highcpu-8` instance (8 vCPUs, 15 GB RAM, GCC 14, `-std=c++23 -O3 -march=znver5 -pthread -fno-exceptions -fno-rtti`), serving as the authoritative record for all 163 newly certified terms;
- `results/A373813_ALL_LINES.txt`—full line coordinates of each optimal cover for $N = 1$ through 1024;
- `results/B373813.txt`—the two-column sequence file $(N, f(N))$ for direct import into OEIS or plotting tools;
- `index.html`—the self-contained interactive demo (see the callout on p. 4), also deployed at `https://prime-line-cover.vercel.app`; and
- `tools/primecover_frontier_diagnosis.sh`—the calibration script described in the preceding paragraph.

Because the bit-width ceiling is architectural rather than algorithmic, extending the record beyond $N = 1024$ requires widening the coverage masks to 2048 bits (thirty-two 64-bit words). The incremental sweep architecture, the Exclusive Dependency Rule, and the parallel frontier mechanism carry over unchanged; the principal costs are the growth of the heavy-line enumeration and the per-task memory footprint, both of which scale with the new horizon. A natural successor is `primecover2048.cpp`.

Acknowledgements

The author's attention was drawn to this problem by a Numberphile video [7] featuring Neil Sloane, who named the rare indices N at which $f(N)$ strictly exceeds $f(N-1)$ *awkward primes*: almost every prime falls onto a line already present in an optimal cover, but the awkward ones land on none of them, forcing the cover to grow. Brady Haran subsequently requested that the sequence be added to the OEIS, where it appears as A393445 [4].

The video lingers on the longest plateau in the staircase—the extended flat run at $f = 69$ visible in figure 7—which spans 112 primes in total ($N = 464$ through $N = 575$). The plateau

opens when $N = 464$ forces the cover to grow to 69 lines; the remaining 111 primes on the plateau are all non-awkward, each landing on an existing cover line, until $p_{576} = 4211$ falls on none of them and forces f to 70. Haran singles out $p_{576} = 4211$ as the *party-pooper prime*. As a small footnote on the efficiency of witness propagation: those 111 non-awkward primes were certified in 111 milliseconds total—one millisecond per prime.

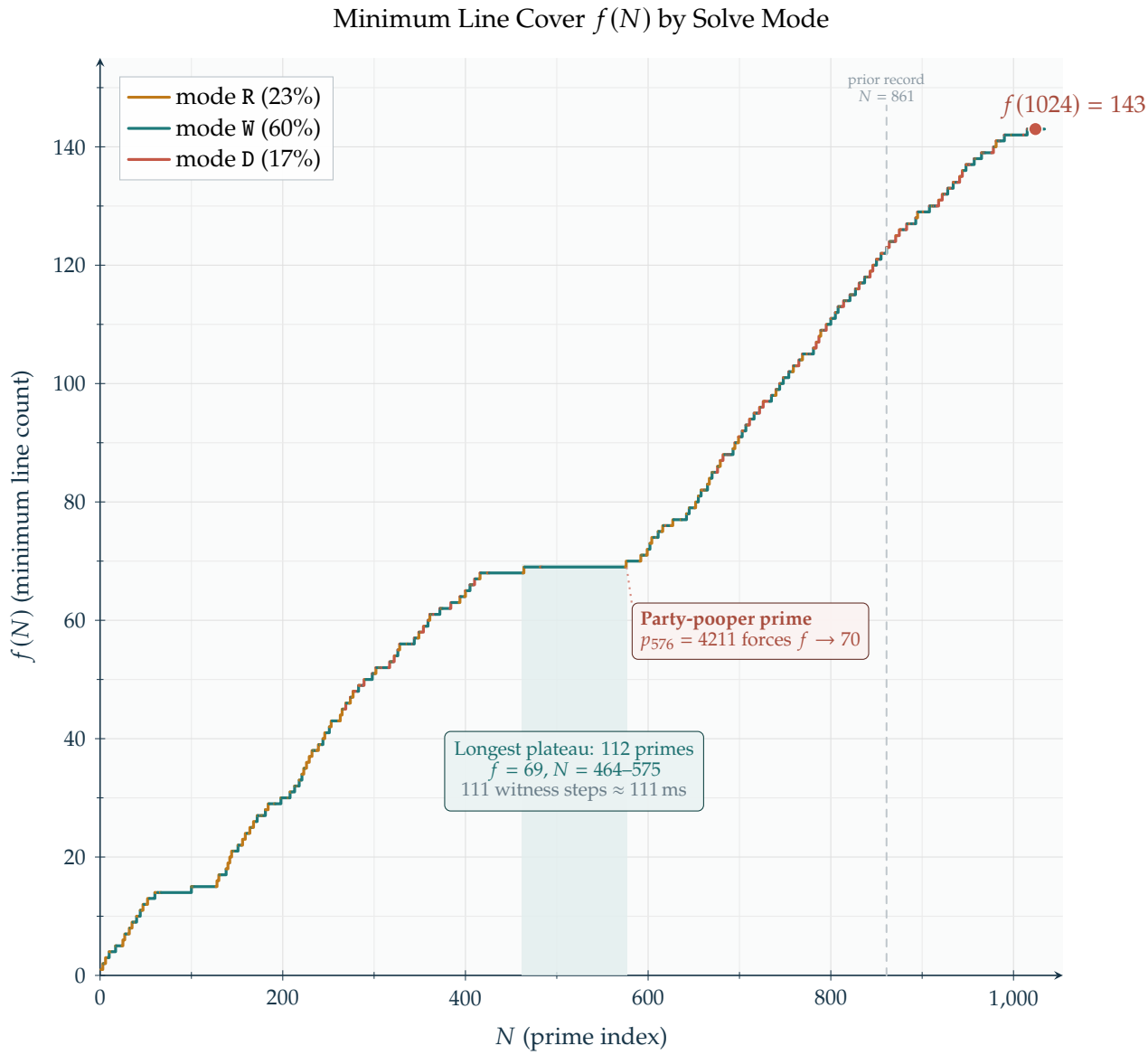

Figure 8: Step-function of $f(N)$ (minimum line count) coloured by solve mode, $N = 1, \ldots, 1024$. The shaded region marks the longest plateau: $f = 69$ holds for $N = 464$–575 (112 consecutive primes), with all 111 non-awkward steps certified by witness bypass in ≈ 111 ms total; $p_{576} = 4211$ (the *party-pooper prime*) then forces $f \to 70$. Colours reflect how optimality was certified at each step: mode `R` (root closure, 23%) closes the gap via Lagrangian bounds at the root without branching; mode `W` (witness bypass, 60%) re-uses a forward witness cover from the preceding step; mode `D` (branch-and-bound, 17%) runs the full parallel frontier search and accounts for virtually all wall-clock time.

References

[1] N. Megiddo and A. Tamir, *On the complexity of locating linear facilities in the plane*, Operations Research Letters, 1(5):194–197, 1982. https://doi.org/10.1016/0167-6377(82)

90039-6

[2] M. A. Alekseyev, *Sage program for lines covering points*, GitHub, August 2024. https://github.com/maxale/oeis/. See also OEIS A373813 contributor notes, https://oeis.org/A373813.

[3] OEIS Foundation Inc., *The On-Line Encyclopedia of Integer Sequences*, A373813, 2026. https://oeis.org/A373813

[4] OEIS Foundation Inc., *The On-Line Encyclopedia of Integer Sequences*, A393445, 2026. https://oeis.org/A393445

[5] J. G. Mikkelsen, *prime-line-cover: Exact Minimum Line Cover Solver for Prime Points (source code, results, and interactive demo)*, GitHub, 2026. https://github.com/jespergran98/prime-line-cover. Interactive demo: https://prime-line-cover.vercel.app.

[6] J. G. Mikkelsen, *prime-line-cover: Exact Minimum Line Cover Solver for Prime Points*, Zenodo, 2026. https://doi.org/10.5281/zenodo.20096556

[7] B. Haran (producer), *Awkward Primes* (featuring N. Sloane), Numberphile, April 7, 2026. https://www.youtube.com/watch?v=VFoIPlUalRY